VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

SEBEORGANIZACE V ROZSÁHLÝCH DISTRIBUOVANÝCH SYSTÉMECH

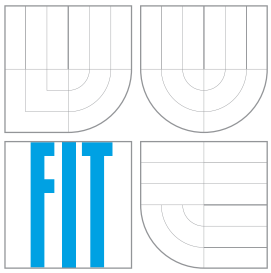
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. MARTIN KUNŠTÁTSKÝ

BRNO 2011

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

SEBEORGANIZACE V ROZSÁHLÝCH DISTRIBUOVANÝCH SYSTÉMECH

SELF-ORGANIZATION IN LARGE DISTRIBUTED SYSTEMS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. MARTIN KUNŠTÁTSKÝ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. SVATOPLUK ŠPERKA

BRNO 2011

Abstrakt

Gossip je generický protokol původně navržený pro šíření informací mezi uzly v rozsáhlých distribuovaných decentralizovaných systémech. Tento protokol lze využít i pro mnoho dalších aplikací včetně agregace dat, konstrukce nejrůznějších topologií, atd. Tato práce popisuje framework určený pro podporu modelování a simulace systémů založených na tomto protokolu.

Abstract

Gossip is a generic protocol which was designed for spreading information between nodes in large distributed decentralised systems. This protocol can be also used for many different applications including data aggregation, topology construction, etc. This work presents and describes a framework designed for facilitating modelling and simulation of Gossip-based systems.

Klíčová slova

Emergence, sebeorganizace, Gossip, self-*, šíření informací, agregace dat, vytváření topologie, peer sampling, distribuované systémy, modelování, simulace, Erlang.

Keywords

Emergence, self-organisation, Gossip, self-*, information dissemination, data aggregation, topology construction, peer sampling, distributed systems, modelling, simulation, Erlang.

Citace

Martin Kunštátský: Sebeorganizace v rozsáhlých distribuovaných systémech, diplomová práce, Brno, FIT VUT v Brně, 2011

Sebeorganizace v rozsáhlých distribuovaných systémech

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Svatopluka Šperky.

.....
Martin Kunštátský
19. května 2011

Poděkování

Tímto bych chtěl poděkovat vedoucímu své práce za odborné vedení, cenné rady a přínosné připomínky.

Obsah

Kapitola 1

Úvod

V současné době bývá v oblasti počítačových systémů stále častěji zmiňován jev, který se dá neformálně nazvat jako *krize komplexnosti*. Počítačové systémy se stávají čím dál tím víc distribuované, neustále se zvětšují a nabývají na složitosti a heterogenitě. Běžně bývá prostřednictvím síťových technologií (ať už drátových či bezdrátových) propojeno velké množství nejrůznějších zařízení nejrůznějších druhů. V rámci počítačových sítí neustále roste objem komunikace. Následkem toho všeho se stává čím dál obtížnější takovéto systémy administrovat.

Jednou z možností, jak čelit náročnosti správy počítačových systému, je navrhovat tyto systémy takovým způsobem, aby vykazovaly vlastnosti jako schopnost sebekonfigurace (*self-configuration*), samostatné řízení (*self-management*), schopnost sebeopravy (*self-repair*, *self-healing*), sebeoptimalizace (*self-optimization*) a podobně [7]. Pro tyto vlastnosti se v anglicky psané literatuře používá souhrnný termín *self-** (rozhodl jsem se jej nepřekládat) [5].

K dosažení takovýchto vlastností u počítačových systémů se nabízí při jejich návrhu využít fenoménů *emergence* a *sebeorganizace* [7][5]. Oba fenomény se běžně vyskytují v přírodě a u nejrůznějších fyzikálních a chemických dějů. Následkem toho upoutaly pozornost vědců z mnoha oborů včetně biologie, fyziky a chemie, ale značný ohlas měly tyto fenomény například i ve filosofii.

Jako příklady emergence a sebeorganizace se uvádí např. komplikované chování mozku živočichů vznikající na základě velice jednoduché interakce neuronů, vznik feromonových cest mravenců při jejich shánění potravy [7], pohyb ptáků v hejnech [10] atp.

V oblasti informačních technologií jsou fenomény emergence a sebeorganizace využitelné zejména při návrhu komplexních systémů. Ačkoli ani jeden z obou termínů není v literatuře chápán zcela konzistentně, je zřejmé, že systémy využívající emergenci a sebeorganizaci mají množství z hlediska návrhu systémů zajímavých vlastností, jako je značná odolnost vůči chybám, schopnost se do jisté míry přizpůsobovat změnám prostředí nebo samostatné řízení [7][5].

Jedním z prostředků využitelných pro návrh systémů využívajících emergenci a sebeorganizaci je protokol Gossip. Ačkoli přesná charakterizace Gossipu není v literatuře zcela ustálena, obecně se jedná o prostředek pro návrh rozsáhlých distribuovaných decentralizovaných systémů, ve kterých jsou si všechny uzly rovny a které pracují s jistou mírou náhodnosti typicky při výběru komunikačního partnera.

Cílem této práce je vytvoření frameworku usnadňujícího modelování a simulaci systémů postavených na protokolu Gossip a tedy vykazujících emergentní a sebeorganizující chování.

Kapitola 2

Emergence a sebeorganizace

Termíny *emergence* a *sebeorganizace* bohužel stále nemají obecně uznávané a používané formálnější definice. Jednotliví autoři k těmto pojmům přistupují různým způsobem, někdy je chápou jako dva zcela rozdílné a na sobě nezávislé pojmy [10][9], jindy je jeden pojem speciální případ či vlastnost doprovázející druhý [7][14], nebo mohou být chápány jako naprosto ekvivalentní [7]. Ke zjednodušení situace navíc nepřispívá ani skutečnost, že se s těmito termíny pracuje ve velkém množství nejrůznějších oborů.

Z tohoto velkého množství různých pojetí jsem se rozhodl vycházet z pohledu, který navrhli autoři Tom De Wolf a Tom Holvoet ve svém článku *Emergence and Self-Organisation: a statement of similarities and differences* ([10]). Přístup autorů k definici pojmů sebeorganizace a emergence je velice systematický a komplexní, na druhou stranu je velice srozumitelný (ve srovnání například s pohledy, které jsou z hlediska této práce sice relevantní, ale orientované mnohem filosofičtěji). Autoři se totiž dlouhodobě zabývají oblastmi relativně blízkými tématu této práce – konkrétně návrhem komplexních systémů multiagentního charakteru.

Zásadní vlastností tohoto pojetí je, že chápe pojmy sebeorganizace a emergence jako na sobě vzájemně zcela nezávislé. Autoři ukazují, že podle jejich definic se každý z těchto fenoménů může vyskytovat samostatně (i když je jejich společný výskyt velice častý).

Možnou nevýhodou pojetí těchto autorů je, že termín sebeorganizace chápou mírně netradičním způsobem. Nicméně autoři ve své práci ukazují, že jejich přístup je ve srovnání s tradičnějšími přístupy výrazně logičtější a systematičtější.

2.1 Emergence

Emergence se obvykle intuitivně chápe jako fenomén, kdy globální chování či vlastnost systému vzniká z interakcí na úrovni lokální, tj. mezi jednotlivými částmi daného systému. Tato globální vlastnost není snadno predikovatelná z vlastností jednotlivých komponent systému a z pohledu redukcionismu se může jevit jako překvapující či neočekávaná [10][14].

De Wolf a Holvoet definují *emergenci* jako vlastnost systému, u kterého existují na globální úrovni (*macro-level*) koherentní emergenty, které dynamicky vznikají z interakcí mezi jednotlivými částmi daného systému na lokální úrovni (*micro-level*). Tyto emergenty jsou nové vzhledem k jednotlivým částem systému.

Termínem *emergent* se chápe libovolný pozorovatelný výsledek emergence, např. chování, vlastnost, struktura atp. Globální úrovní se chápe úhel pohledu takový, kdy se na systém díváme jako na celek. Proti tomu lokální úroveň chápeme jako pohled na jednotlivé

prvky systému.

Autoři svou definici doplňují seznamem charakteristických znaků emergence. Za nejpodstatnější vlastnost považují *mikro-makro efekt* (*micro-macro effect*), tj. skutečnost, kdy vlastnosti, chování, struktury nebo vzory umístěné na globální úrovni vznikají v důsledku interakcí mezi jednotlivými částmi systému, tj. na lokální úrovni. Takto vzniklé emergenty zase zpětně ovlivňují lokální úroveň – dochází tak k *obousměrnému propojení* (*two-way link*) mezi lokální a globální úrovní.

Další podle autorů velice důležitou vlastností je *radikální novost* (*radical novelty*) globálního chování vzhledem k chování jednotlivých částí systému. Jednotlivé části systému nemají explicitní reprezentaci globálního chování. Nicméně globální chování je pořád implicitně obsaženo v chováních jednotlivých částí systému a může být pochopeno a predikováno, zkoumáme-li chování těchto částí s ohledem na chování zbytku systému (je potřeba brát v úvahu chování ostatních částí a jejich vzájemné propojení). Nicméně toto chování nebude predikovatelné, pokud se díváme pouze na jednotlivé části systému bez ohledu na celek.

Důsledkem toho, že žádná část systému nemá explicitní reprezentaci celku, je, že systémy s emergentním chováním mají *decentralizované řízení* (*decentralised control*). Neexistuje prvek systému, který by sám určoval chování celku, chování celku je důsledek lokálních interakcí. Následkem toho je, že systém je velice *robustní* – selhání jedné části systému nemůže způsobit selhání celku (i když samozřejmě určitý vliv na výkonnost celého systému mít může).

Mezi další podstatné vlastnosti patří skutečnost, že emergenty vznikají *dynamicky* v průběhu vývoje systému v čase, nejsou tedy dány nijak od začátku. Emergenty vznikají v důsledku *interakce* jednotlivých částí systému mezi sebou.

Poslední podstatnou vlastností emergentů zmiňovanou De Wolfem a Holvoetem je *koherence* – emergenty jsou integrované celky, které si udržují jistou identitu.

2.2 Sebeorganizace

Sebeorganizaci, tak jak ji definovali De Wolf a Holvoet, můžeme intuitivně chápat jako schopnost systému organizovat sám sebe (tj. modifikovat svou strukturu) bez nějakého řízení z vnějšku [10][9]. Nicméně mnozí jiní autoři chápou sebeorganizaci jiným způsobem [7][14].

Definice pojmu *sebeorganizace* (*self-organisation*) podle De Wolfa a Holvoeta: Sebeorganizace je dynamický a adaptivní proces, kde systém získává a udržuje svou vlastní strukturu sám, bez nějakého vnějšího řízení.

Organizací systému se chápe uspořádání jeho jednotlivých prvků tak, aby byla umožněna určitá funkčnost celku. Aby byl systém schopen vykonávat určitou užitečnou funkci, nesmí být jeho míra uspořádání ani příliš malá ani příliš velká [10]. Sebeorganizující se systém je tedy systém, který sám udržuje svou míru organizace na takové úrovni, aby byl schopen vykonávat nějakou činnost.

Mezi charakteristické vlastnosti systémů se sebeorganizujícím se chováním, které autoři uvádějí, patří *vzrůst řádu* (*increase in order*) chování systému, díky kterému systém získá prostorovou, časovou nebo funkční strukturu. Počáteční stav systému je zcela neuspořádaný nebo jen částečně uspořádaný, míra uspořádání se postupně zvyšuje. K nárůstu uspořádání dochází v průběhu časového vývoje systému – sebeorganizace je *dynamický proces*.

Další podstatnou vlastností sebeorganizujících se systémů je jejich *autonomie* – systém je schopen organizovat svou vnitřní strukturu bez zásahu z vnějšku, bez jakéhokoli vnějšího řízení. Neexistence vnějšího řízení samozřejmě neznamená, že by systém neměl mít žádné

vstupy. Samoorganizující se systém musí být otevřený, nicméně by se měl sám rozhodovat, jakým způsobem bude na dané vstupy reagovat. Vstupní data by tedy neměla mít řídící charakter.

Pro určení, jestli je systém sebeorganizující se, je potřeba důkladně vymezit hranice systému. Sebeorganizující se systém (podle předchozí definice) může mít řídící prvek, který určuje chování celku. Pokud tento řídící prvek vyjmeme z definice systému, tj. povedeme hranici systému tak, že tento řídící prvek zůstane vně systému (bude tedy součástí vnějšího prostředí), nebude se již jednat o sebeorganizující se systém.

Autoři dále zmiňují jako podstatnou vlastnost sebeorganizujících se systémů jejich *adaptabilitu* na změny prostředí. Sebeorganizující se systém by se měl být schopen vyrovnávat se změnami prostředí, měl by vykazovat schopnost adaptace na změny. K tomu je nutné, aby systém byl schopen generovat velké množství různých chování a aby vybíral vhodné chování podle aktuálního vstupu.

Sebeorganizující se systém je dynamický nelineární systém s atraktorem, jehož třída je na hraně chaosu. Atraktor typu pevný bod nebo periodický atraktor by kladly příliš velká omezení na možnosti chování systému (pevný bod umožní pouze jediný typ chování, periodický atraktor vede k periodickému chování), naopak chaotický atraktor by způsobil, že se systém začne chovat nepředvídatelně – bude vykazovat příliš velké množství různých chování, z hlediska pozorovatele bude jeho chování vypadat jako zcela náhodné. Sebeorganizující se systém by tedy měl vykazovat chování na hranici těchto dvou extrémů – na hraně chaosu. To umožní systému chovat se dostatečně dynamicky, aby byl schopen reagovat na změny prostředí, ale zároveň jeho chování nebude sebedestruktivně chaotické.

Sebeorganizace nemusí nutně znamenat, že systém bude robustní ve smyslu schopnosti se vyrovnat s poškozením či poruchou jedné ze svých částí. Tato vlastnost je v pojetí De Wolfa a Holvoeta typická pro emergenci ale nikoli nutně pro sebeorganizaci.

Luís Correia ve své práci *Self-organised systems: fundamental properties* ([9]) v podstatě opakuje předchozí De Wolfovu a Holvoetovu definici a popis sebeorganizace, nicméně je doplňuje o některé další zajímavé vlastnosti. Jednak zdůrazňuje nutnost *interakce* mezi jednotlivými prvky sebeorganizujícího se systému (tato vlastnost není v De Wolfově a Holvoetově práci zmíněna explicitně, avšak z ní přímo vyplývá). Correia tvrdí, že jsou možné dva typy interakce: *kooperativní* a *kompetitivní*. Kooperativní interakce je zcela přirozená, ke kompetitivní interakci dochází tehdy, jsou-li jednotlivé prvky systému nuceny sdílet zdroje.

Další vlastností sebeorganizujících se systémů, kterou Correia zmiňuje, je *asynchronnost* mezi jednotlivými prvky systému. Ve fyzických systémech neexistuje (dokonalá) synchronizace – synchronní chování je možné pouze od určité úrovně abstrakce. U fyzických systémů se signál vždy šíří konečnou rychlostí. Tudíž se vnější podnět může dostat k různým prvkům systému s různým zpožděním. Každý prvek tedy může reagovat na podnět v různou dobu. Fyzický sebeorganizující se systém tedy nemůže spoléhat na dokonalou synchronnost svých komponent, je tedy asynchronní.

2.3 Vztah sebeorganizace a emergence

Sebeorganizace a emergence jsou fenomény, které jsou v literatuře často diskutovány společně. Většina tradičně uváděných příkladů na sebeorganizaci či na emergenci jsou případy systémů, u kterých se objevuje kombinace obou těchto fenoménů. To je pravděpodobně také jednou z příčin určitého chaosu v definicích těchto fenoménů [10].

Nicméně De Wolf a Holvoet uvádějí případy, kdy je přítomen pouze jeden z těchto dvou fenoménů. Příkladem sebeorganizujícího se systému bez přítomné emergence je libovolný

(například multiagentní) systém, kde jeden prvek má explicitní model výsledného chování a tento prvek přímo řídí chování ostatních prvků tak, aby celek vykazoval samoorganizující chování [10]. Takovéto uspořádání bude pravděpodobně velice běžné, bude-li systém s požadovaným samoorganizujícím chováním navržen tradičními inženýrskými metodami.

Příkladem emergence bez přítomné samoorganizace může být téměř jakákoli vlastnost stabilní tekutiny (objem, tlak, ...). Tato vlastnost vzniká interakcí mezi jednotlivými částicemi tekutiny a zase zpětně tyto částice ovlivňuje. Nicméně pokud se tekutina nachází ve stabilním stavu, nelze mluvit o sebeorganizaci (sebeorganizující se systém musí být ve stavu vzdáleném od ekvilibria) [10].

Případy systémů s emergentním chováním nedoprovázeným sebeorganizací jsou překvapivě časté. Jedná se o systémy v rovnovážném stavu nebo blízko něj, jejichž globální vlastnosti jsou odvozeny od interakcí mezi jednotlivými částmi. V podstatě všechny globální vlastnosti libovolného stabilního fyzického hmotného objektu (ať už pevného, kapalného či plynného) mají takovýto charakter [14]. Tyto vlastnosti jsou obvykle snadno predikovatelné, obvykle je lze snadněji modelovat na globální úrovni než na úrovni jednotlivých prvků systému (tj. na molekulární či atomické úrovni) [14].

Kombinace emergence a sebeorganizace je možná pouze v nelineárních systémech daleko od rovnovážného stavu. Jedná se o otevřené systémy, které se udržují mimo rovnovážný stav díky přísunu energie či hmoty z vnějšího prostředí [14]. Tyto systémy sice generují entropii, ale ta je aktivně ze systému aktivně odváděna. Jedná se tedy o *disipativní systémy*.

De Wolf a Holvoet zmiňují jednu podstatnou vlastnost systému kombinujících sebeorganizaci a emergenci: *nelinearitu* jejich chování. Tato vlastnost bývá někdy v literatuře uváděna jako vlastnost buď emergence nebo sebeorganizace, jedná se však spíše o vlastnost kombinace těchto dvou fenoménů. Nelineárního chování je dosahováno díky pozitivním zpětným vazbám, které zesilují či násobí následky počáteční relativně malé příčiny – díky nim vznikají v dynamických systémech emergentní vlastnosti.

Množství známých příkladů na kombinaci sebeorganizace a emergence se týká například eusociálního hmyzu jako jsou mravenci či včely. V literatuře bývají často uváděny jako příklady kombinace těchto fenoménů například vznik a sebeoptimalizace globálních feromonových cest mravenců [10] nebo seskupování mravenčích kukel dělnicemi [10][7].

2.4 Silná a slabá emergence

Vedle fenoménu emergence, jak byl popsán dříve, existuje také koncept *silné emergence* (*strong emergence*). Silná emergence popisuje případ, kdy se vlastnost systému jako celku nedá vůbec nijak odvodit od vlastností jednotlivých částí [7][6]. Termín se používá spíše jen ve filosofii, v oblasti návrhu systémů nemá žádné uplatnění. Navíc existují pochyby, jestli vůbec takovýto fenomén ve fyzickém světě existuje – to, že se nám jistá vlastnost systému jeví jako projev silné emergence, může být pouhým důsledkem nedostatku informací o něm [7].

Je-li potřeba brát v úvahu i silnou emergenci, používá se pro koncept emergence popsaný v části 2.1 termín *slabá emergence* (*weak emergence*). Pokud v této práci mluvím o emergenci, aniž bych specifikoval, zda se jedná o slabou či silnou emergenci, vždy mám na mysli emergenci slabou.

2.5 Využití kombinace emergence a sebeorganizace pro návrh systémů

Kombinace emergence a sebeorganizace je vhodná pro návrh určitého typu systémů.

Často potřebujeme sestavit komplexní systém (např. multiagentního charakteru) z velkého množství relativně jednoduchých entit, nicméně požadované chování by mělo být dostatečně komplexní. Jeden jednoduchý prvek však nedokáže řídit takovéto chování celku. Nabízí se proto navrhnout systém tak, aby chování celku vzniklo emergentně z interakcí mezi chováním jednotlivých prvků. Nicméně toto u komplexního chování není možné bez sebeorganizace (emergence bez sebeorganizace je možná pouze u stabilních systémů). Systém tedy musí vykazovat jak sebeorganizaci tak emergenci [10].

Požadované vlastnosti naprosto samostatných systémů, tj. sebekonfigurace, sebeoprava, sebeoptimalizace a samostatné řízení (*self-** vlastnosti) jsou uváděny v literatuře jako vlastnosti sebeorganizace [7]. Nicméně v pojetí, které používám v této práci, se jedná o vlastnosti systémů, které kombinují sebeorganizaci a emergenci. Chci-li u navrhovaného systému dosáhnout *self-** vlastností, mohu s úspěchem využít kombinace emergence a sebeorganizace.

Kapitola 3

Gossip

Gossip je generický protokol původně navržený pro distribuci informací v rozsáhlých decentralizovaných distribuovaných systémech [5]. V tomto pojetí, tedy jako randomizovaný algoritmus pro šíření informací, se poprvé objevil v roce 1987 v práci *Epidemic Alhorithms for Replicated Database Maintenance* (kolektiv autorů: A. Demers, D. Greene, et al.; [11]). Později byl koncept vycházející z této práce zobecněn a začal být využíván i pro další účely. Dnes se Gossip používá pro velké množství nejrůznějších aplikací [5].

Gossipové protokoly jsou určeny pro rozsáhlé distribuované systémy, kde jsou si všechny prvky (*uzly*) rovny v tom smyslu, že žádný není nijak zvýhodněn či znevýhodněn vůči ostatním. Každý uzel periodicky spouští relativně jednoduchý kód, který je pro všechny uzly stejný. Tento kód zahrnuje komunikaci s jedním dalším uzlem a možnou změnu stavu uzlu v důsledku této komunikace. Uzly komunikují výhradně po dvojicích, komunikace typu broadcast je možná až jako aplikace těchto protokolů.

K výběru partnerského uzlu pro komunikaci dochází buď náhodně z celé množiny zbývajících uzlů, nebo uzel komunikuje s několika sousedy danými aktuální topologií, která však nemusí být neměnná. Některé aplikace vyžadují dokonale rovnoměrný náhodný výběr komunikačního partnera, jindy je potřeba náhodný výběr určitým způsobem zvýhodňující některé uzly, atp.

Ačkoli to nebývá v literatuře zdůrazněno, neexistuje mezi uzly žádná globální synchronizace – uzly tedy pracují paralelně a asynchronně. Některé aplikace však vyžadují, aby byly všechny uzly spouštěny z dlouhodobého pohledu stejně často, tedy aby žádný uzel nespouštěl svůj kód výrazně častěji nebo výrazně méně často než uzly zbývající [16].

Systémy využívající gossipový protokol je potřeba navrhovat tak, aby u nich vznikaly požadované globální vlastnosti emergentně v důsledku pravidelně opakované lokální komunikace mezi jednotlivými uzly [5]. Takto navržené systémy potom mohou mít a využívat vlastnosti systémů kombinujících emergenci a sebeorganizaci, tj. robustnost, určitá odolnost vůči selhání jednotlivých prvků, schopnost se do jisté míry přizpůsobovat změnám okolního prostředí, schopnost autonomního řízení, absence kritických centrálních řídících prvků, atd. – tedy *self-** vlastnosti.

3.1 Kostra gossipového protokolu

V každém uzlu běží dvě výpočetní vlákna – vlákno *aktivní* a vlákno *pasivní*. Jejich chování je možné chápat jako chování klienta (aktivní) a serveru (pasivní). Klientské vlákno uzlu aktivně a opakovaně v určitých časových intervalech navazuje komunikaci s jinými uzly,

serverové vlákno reaguje na navázání komunikace z cizího uzlu [5][4].

Podle směru šíření informace rozlišujeme tři varianty Gossipu: *push*, *pull* a *push-pull*. U *push* varianty se informace při komunikaci šíří ve směru od uzlu, který komunikaci navázal; u *pull* varianty je tomu naopak – uzel aktivně navazující komunikaci informaci obdrží. V případě *push-pull* varianty dojde po navázání komunikace k výměně informací oběma směry.

Pseudokód aktivního vlákna varianty *push-pull*: [5]

```
1: while 1
2:     wait(t)
3:     peer := selectPeer()
4:     send state to peer
5:     receive state_peer from peer
6:     state := update(state, state_peer)
7: end
```

Vynecháním řádků 5 a 6 vznikne *push* varianta, obdobně při vynechání řádku 4 získáme variantu *pull*. Aktivní vlákno provádí svou činnost v pravidelných časových intervalech t. Funkce selectPeer() vrací identifikátor cizího uzlu *peer*, se kterým hodlá vlákno komunikovat. V případě *push* nebo *push-pull* varianty tomuto uzlu pošle svůj vlastní stav *state*. V případě variant *pull* nebo opět *push-pull* obdrží od tohoto uzlu jeho stav $state_{peer}$. Následně funkce update() nastaví nový stav uzlu na základě původního stavu a nově získané informace.

Pseudokód pasivního vlákna varianty *push-pull*: [5]

```
1: on receive(peer)
2:     receive state_peer from peer
3:     send state to peer
4:     state := update(state, state_peer)
5: end
```

Vynecháním řádku 3 vznikne *push* varianta, vynecháním řádků 2 a 4 zase varianta *pull*. Kód se spouští na základě zahájení komunikace ze strany cizího uzlu *peer*.

Toto obecné schéma obsahuje následující aplikačně závislé komponenty:

- (lokální) stav *state* uzlu
- funkci update(), vracející hodnotu nového stavu na základě stavu minulého a stavu získaného od komunikačního partnera
- funkci selectPeer(), která vrací partnera pro komunikaci

Při vytváření konkrétní aplikace protokolu Gossip musejí být specifikovány tyto tři komponenty a použitá varianta protokolu [4][15].

Způsob inicializace jednotlivých uzlů Gossipu není nijak obecně specifikován. Inicializace je závislá jednak na vlastnostech dané aplikace, jednak na konkrétní implementaci Gossipu.

V pseudokódu dále není definováno, jestli komunikace uzlů má být implementována jako synchronní či asynchronní. V případě synchronní komunikace není mezi implementacemi jednotlivých variant zásadní rozdíl – liší se pouze ve směru přenášené informace.

Avšak v případě asynchronní komunikace se varianta *push* podstatně odlišuje od ostatních dvou variant tím, že její aktivní vlákno nemusí čekat na odpověď od pasivního vlákna komunikačního partnera. U varianty *push* tedy nemůže dojít k blokování aktivního vlákna.

Varianty *pull* a *push-pull* se i v případě asynchronní komunikace chovají synchronně (v důsledku komunikace prostřednictvím dotazu a odpovědi dochází k synchronizaci aktivního vlákna uzlu, který vyvolal komunikaci, s pasivním vláknem jeho komunikačního partnera). Aktivní vlákno uzlu tedy bude po odeslání zprávy zablokováno, dokud neobdrží od pasivního vlákna komunikačního partnera odpověď.

Při implementaci je potřeba počítat s tím, že v jakémkoli uzlu může kdykoli dojít k poruše – tudíž musíme počítat i s případem, kdy dojde k výpadku uzlu, s jehož pasivním vláknem byla právě navázána komunikace. Proto je potřeba zajistit, aby aktivní vlákno jiného uzlu nemohlo být v důsledku takovéhoto výpadku jeho komunikačního partnera zablokováno trvale – tedy zajistit, aby aktivní vlákno čekalo na odpověď pouze omezenou dobu a v případě neobdržení odpovědi v daném časovém limitu bylo schopno přeskočit v daném cyklu provedení funkce update() a pokračovat další iterací.

Ačkoli na jednotlivých uzlech probíhá výpočet asynchronně, je v literatuře běžné analyzovat běh aplikace Gossipu prostřednictvím časových intervalů zvaných cykly (*cycles*) nebo kola (*rounds*). Jednotlivé definice uváděné v literatuře se vzájemně liší (např. [21], [5]). Zde budu používat termín *cyklus* pro časový interval, během kterého dojde k provedení průměrně jednoho cyklu aktivního vlákna u každého uzlu.

Je zřejmé, že délka cyklu je určena v první řadě funkcí wait(), ačkoli je v případě *pull* nebo *push-pull* varianty (a při použití synchronní komunikace i u varianty *push*) ovlivněna (při nevhodně nastavených parametrech systému i dosti výrazně) dobou čekání na odpověď od komunikačního partnera.

U variant *push* a *pull* dojde pro každý uzel k jednomu přenosu stavu za jeden cyklus. U varianty *push-pull* však dojde během jednoho cyklu ke dvěma takovýmto přenosům. Dá se tedy předpokládat (ačkoli je to zřejmě závislé na konkrétní aplikaci), že systémy využívající *push-pull* komunikace se budou vyvíjet rychleji, než systémy založené na *push* nebo *pull* Gossipu.

V literatuře lze nalézt množství aplikací, které jsou sice označené jako gossipové, avšak kód jejich uzlů nelze nijak jednoduše vložit do obecné kostry Gossipu popsaného v této části (příklady takovýchto aplikací např. [18][17][13][5]). Tyto aplikace mají množství vlastností společných se zde popsaným pojetím: vždy pracují nad rozsáhlými distribuovanými systémy, ve kterých jsou si všechny uzly rovny, typický je náhodnostní přístup (náhodný výběr komunikačního partnera, náhodný výběr provedené operace, atp.), každý uzel spouští opakovaně relativně jednoduchý kód, který je pro všechny uzly v podstatě stejný a zahrnuje komunikaci s jedním či s větším množstvím dalších uzlů. Algoritmus těchto aplikací je často možné upravit tak, aby jej bylo možné zapsat ve formě výše zmíněné kostry, nicméně tato úprava bude zcela jistě mít negativní vliv na intuitivnost a čitelnost algoritmu (popis algoritmu bude zbytečně komplikovaný) a může mít i určitý vliv na efektivitu nebo dokonce i rychlost algoritmu.

V této práci tedy budu považovat za gossipové aplikace všechny, které výše zmíněné kostře přesně odpovídají nebo které se od této kostry liší jen relativně nepatrně. Algoritmy aplikací, jejichž popis výše zmíněné kostře zcela přesně neodpovídá, budu upravovat do tvaru této kostry pouze tehdy, neovlivní-li to nijak zásadně jejich výkonnost.

3.2 Modularita Gossipu

Vzhledem k tomu, že požadovaná funkčnost vzniká u aplikací Gossipu emergentně, je návrh aplikací Gossipu vykonávajících nějakou komplexnější činnost dosti náročný.

Proto se objevila myšlenka využít při návrhu aplikací Gossipu modulárního paradigmatu [20]. Aplikace s komplexnějším chováním bude sestavena z většího množství relativně jednoduchých komponent. Každá takováto komponenta je představována aplikací Gossipu vykonávající nějakou jednoduchou a snadno pochopitelnou činnost. Komponenty mají pevně definované rozhraní, které umožňuje jejich vzájemnou komunikaci.

Tento přístup má řadu výhod – kromě zjednodušení návrhu dekompozicí komplexního problému na řadu jednodušších podproblémů umožňuje dále opakované využití již existujících komponent, snadnější udržovatelnost, atp.

Typická jednodušší gossipová aplikace se bude skládat ze dvou komponent:

- komponenta zajišťující sestavení a udržování komunikační topologie
- funkční komponenta vykonávající nějakou užitečnou činnost

Samozřejmě je možný i podstatně komplikovanější návrh s mnohem větším množstvím komponent různých typů.

Komponenty pro sestavování a udržování komunikační topologie typicky slouží pro zajištění funkce selectPeer() s požadovanou funkčností jak pro sebe, tak i pro funkční komponentu. Takovéto komponenty budou podrobněji popsány v podkapitolách 4.1 a 4.2.

V systému navrhovaném s využitím modularity tedy poběží na každém uzlu nezávisle několik aplikací Gossipu. Aplikace tvoří hierarchii, ve které nižší vrstvy poskytují určité služby vrstvám vyšším. V takovémto systému existují dva druhy komunikace [20]:

- v rámci jednoho uzlu mezi různými aplikacemi (meziaplikační komunikace)
- mezi různými uzly v rámci jedné aplikace (meziuzlová komunikace)

Aplikace dále komunikují s vnějším prostředím – s uživateli aplikace, s různými senzory poskytujícími data pro aplikaci, atp.

Kapitola 4

Aplikace protokolu Gossip

Protokol Gossip lze využít pro velké množství nejrůznějších aplikací. Zde uvádím pouze několik v literatuře často zmiňovaných příkladů.

Jak už bylo uvedeno v předchozí kapitole, je možné rozdělit aplikace Gossipu na dvě skupiny. Vedle funkčních aplikací provádějících přímo nějaký užitečný výpočet (agregace, šíření informací, atp.) jsou tu ještě aplikace pro zajištění komunikačních topologií. Tato druhá skupina aplikací slouží pouze jako prostředek pro zajištění služeb pro podporu běhu aplikací funkčních.

U jednotlivých aplikací je také doplněn popis možné implementace dané aplikace jako demonstrační úlohy. U aplikací odpovídajících obecné kostře gossipové aplikace jsou specifikovány všechny aplikačně závislé komponenty Gossipu, tedy forma stavu uzlu, použitá varianta protokolu (*push*, *pull* nebo *push-pull*) a definice funkcí update() a selectPeer().

4.1 Vytváření topologie

Množství aplikací pro svůj běh vyžaduje, aby jednotlivé prvky systému byly uspořádané do určité stabilní komunikační topologie. Příkladem takovýchto aplikací může být například směrování paketů [15], řazení číselných hodnot [16], vyhledávání v uspořádaných množinách [16], atp.

V rozsáhlých decentralizovaných distribuovaných systémech může být vytváření a udržování takovéto topologie dosti netriviální problém, zvláště pokud může za běhu systému docházet k zániku existujících uzlů a vzniku nových.

Komunikační topologii můžeme popsat *orientovaným grafem* $G = (V, H)$, kde množina uzlů V tohoto grafu je totožná s množinou uzlů systému a množina hran H je definována tak, že každému uzlu x je přiřazena množina uzlů N_x, $x \notin N_x$, taková, že $i \in N_x \Leftrightarrow (x, i) \in H$, tedy (x, i) je hrana v orientovaném grafu pro všechny prvky $i \in N_x$. Množina uzlů N_x (sousední uzly uzlu x) pak určuje uzly, se kterými daný uzel x komunikuje. Tato množina se nazývá pohled (*view*) uzlu x. Takto definovaný graf určuje *překryvovou síť* (*overlay network*) nad daným systémem.

Předpokládejme, že každý uzel je jednoznačně identifikován svou adresou a že znalost adresy je nutná a postačující znalost pro komunikaci s daným uzlem.

Předpokládejme dále, že nám Gossip poskytuje službu, danou například funkcí selectPeer(), která danému uzlu vrátí adresu jednoho zcela náhodného prvku z celé množiny zbývajících uzlů. V tomto případě může každý uzel potenciálně komunikovat s kterýmkoli jiným. Nicméně mu tento mechanismus neumožní komunikovat (kdykoli potřebuje) s nějа-

kým jedním konkrétním uzlem. Chceme-li uzlům umožnit, aby si samy mohly určit, se kterými uzly budou komunikovat, musíme jim dát možnost zapamatovat si adresy daných uzlů (množina adres uzlů, se kterými může uzel komunikovat, se tak stane součástí jeho stavu) a možnost ovlivňovat funkci selectPeer() [15].

Márk Jelasity a Ozalp Babaoglu navrhli ve své práci *T-Man: Fast Gossip-based Construction of Large-Scale Overlay Topologies* ([16]) gossipový protokol nazvaný *T-Man*, určený pro sestavování topologií nad rozsáhlými distribuovanými systémy. Protokol využívá seřazovací funkci (*ranking function*) R, která je schopna pro zadaný libovolný uzel x seřadit vstupní množinu uzlů podle jejich míry vhodnosti stát se sousedním uzlem daného uzlu x. Míra vhodnosti dvou uzlů stát se sousedními uzly je dána konkrétní aplikací, pro kterou je topologie sestavována, může být určena například podle fyzické vzdálenosti daných uzlů (fyzicky blízký uzel je jako soused mnohem vhodnější než uzel fyzicky vzdálenější), atp.

Autoři ve výše zmíněné práci definují problém konstrukce topologie: Mějme množinu uzlů V o velikosti $|V| = n$, konstantu c určující požadovaný počet sousedů každého uzlu ve výsledné topologii (tj. $c = |N_x|$ je velikost pohledu každého uzlu x) a řadící funkci R, která pro daný uzel x a množinu uzlů A, $x \notin A$, vrací částečné uspořádání uzlů v množině A podle jejich míry vhodnosti stát se sousedním uzlem uzlu x. Částečné uspořádání je reprezentováno ve formě množiny několika uspořádání úplných, $R(x, A)$ je tedy množina několika úplných uspořádání množiny A. Řadící funkci R je vhodné definovat pomocí funkce $d(x, y)$ definující metrický prostor nad množinou uzlů V [16], tj. funkce $d(x, y)$ určuje vzdálenost uzlů x a y.

V základní verzi spočívá problém konstrukce topologie v konstrukci pohledů všech jednotlivých uzlů tak, aby pohled N_x uzlu x obsahoval přesně prvních c prvků z některého úplného uspořádání všech zbývajících uzlů podle jejich vhodnosti stát se sousedy uzlu x. Našim cílem tedy je pro každý uzel x najít takové N_x, aby obsahovalo přesně prvních c uzlů z libovolného jednoho úplného uspořádání z množiny $R(x, V - \{x\})$.

Autoři dále definují další variantu předchozího problému – *topology embedding problem* (rozhodl jsem se nepřekládat). V tomto případě značí c velikost pohledu N_x, nicméně požadovaný počet sousedů ve výsledné topologii je $k \leq c$. Cílem v tomto případě je, aby pohled N_x obsahoval prvních k prvků z některého z množiny uspořádání $R(x, V - \{x\})$. Zbývajících $c - k$ prvků množiny N_x může být libovolných. Ve výsledné topologii je pak množina sousedních uzlů uzlu x dána prvními k prvky z některého z uspořádání $R(x, N_x)$. Základní verze problému konstrukce topologie je tedy speciální varianta problému *topology embedding*, ve které platí $k = c$.

Protokol *T-Man* je gossipový protokol sloužící k řešení problému *topology embedding*. Protokol vyžaduje, aby každý uzel měl svůj *deskriptor*, který se skládá z adresy daného uzlu a z jeho *profilu*. Profil uzlu je určen aplikací a je využíván řadící funkcí R k určení částečného uspořádání. Například řadí-li funkce $R(x, A)$ prvky z množiny A podle jejich fyzické vzdálenosti od aktuálního prvku x, bude profil každého uzlu obsahovat souřadnice jeho polohy. Při výpočtu uspořádání se pro každý prvek z množiny A určí jeho euklidovská vzdálenost od uzlu x a dané prvky budou poté podle této vzdálenosti vzestupně seřazeny.

Pokud chceme zachovat obecný tvar protokolu Gossip (autoři jej ve své práci přesně nezachovávají, nicméně jejich popis je na zde použité pojetí snadno převoditelný), definujeme stav uzlu jako dvojici skládající se z aktuálního pohledu uzlu a z jeho profilu. Profil uzlu je neměnný, tedy při vytváření nového stavu se prostě zkopíruje ze stavu starého. Aktuální pohled uzlu je množina deskriptorů uzlů o velikosti c. Na počátku by měl pohled uzlu obsahovat c deskriptorů uzlů vybraných náhodně z celé množiny uzlů. Na konci běhu protokolu tvoří prvních k prvků z některého uspořádání pohledu prostřednictvím funkce R množinu

sousedních uzlů ve výsledné topologii.

Funkce update() provede sjednocení aktuálních pohledů z obou stavů a do takto získané množiny deskriptorů ještě přidá deskriptor uzlu, se kterým právě komunikoval. Dále tuto množinu seřadí pomocí funkce R a vrátí nový stav s pohledem, který se skládá z prvních c prvků z jednoho z takto získaných úplných uspořádání. Jinými slovy: Nechť *state* je aktuální stav uzlu, *state.view* je aktuální pohled (jako součást stavu) a *state.descriptor* je deskriptor uzlu se stavem *state* (deskriptor je pro daný uzel neměnný, nicméně je praktičtější jej považovat za součást stavu). Pak funkce update(*state*, $state_{peer}$) provede sjednocení množin

$$state.view \cup state_{peer}.view \cup \{state_{peer}.descriptor\}$$

a z takto vzniklé množiny deskriptorů použije přesně c nejvhodnějších prvků pro nový pohled (součást nového stavu). Nejvhodnější uzly získá tak, že vezme jedno z úplných uspořádání vrácených funkcí R nad danou množinou a z něj použije prvních c prvků.

Funkce selectPeer() pracuje tak, že nejprve uspořádá (funkcí R) aktuální pohled daného uzlu, a z první poloviny takto získané posloupnosti vybírá jeden náhodný prvek s rovnoměrným rozložením pravděpodobnosti (vybírá tedy mezi $\frac{c}{2}$ prvky). Funkce selectPeer() je tedy závislá na aktuálním stavu uzlu.

Protokol *T-Man* využívá variantu Gossipu *push-pull*.

Autoři ukazují, že systém využívající protokol *T-Man* konverguje k požadované topologii v čase $O(\log n)$, kde n je počet prvků v systému.

4.1.1 Příklad implementace vytváření topologie mřížka

Našim cílem je uspořádat uzly do struktury obecně n-rozměrné mřížky. Označme množinu uzlů jako V. Předpokládejme pro jednoduchost, že $|V| = m^n$, kde m je přirozené číslo. Nechť je každému uzlu i z množiny V přiřazena unikátní n-tice přirozených čísel menších nebo rovných m (tj. z množiny $\{1, 2, \ldots, m\}$). Tato n-tice tvoří *profil* daného uzlu.

Vzdálenost dvou profilů je definována jako Manhattanská vzdálenost. Tedy vzdálenost profilů $\vec{x} = (x_1, x_2, \ldots, x_n)$ a $\vec{y} = (y_1, y_2, \ldots, y_n)$ je definována jako $d(\vec{x}, \vec{y}) = |x_1 - y_1| + |x_2 - y_2| + \cdots + |x_n - y_n|$.

Ve výsledné topologii by každý uzel s výjimkou krajních měl mít přesně $2n$ sousedů. Označme pohled uzlu i jako $view_i$. Pokud by velikost pohledu $c = |view_i|$ byla rovna $2n$, pro nízké hodnoty n (které se dají předpokládat) by hrozilo, že počáteční graf po náhodné inicializaci pohledů bude rozpojený. Proto bude použita velikost pohledu $c > 2n$ (ideálně v řádu desítek) [16]. Jedná se tedy o řešení problému *topology embedding*.

Stav uzlu je definován jako dvojice $(view, descriptor)$, kde *view* je aktuální pohled daného uzlu, tj. množina deskriptorů o velikosti c, a *descriptor* je jeho deskriptor. Deskriptor uzlu je tvořen jeho adresou a profilem. Deskriptor uzlu je sice součástí jeho stavu, nicméně je samozřejmě neměnný. V následujícím pseudokódu budu k pohledu uzlu se stavem *state* přistupovat jako *state.view*, podobně k jeho deskriptoru jako *state.descriptor*.

Funkce update() je definována následovně:

```
function update(state, state_peer)
    set = (state.view ∪ state_peer.view ∪ {state_peer.descriptor}) - state.descriptor
    return (selectFirst(c, sort(set, state.profile)), state.descriptor)
end
```

Funkce sort(s, $profile$) vrací jedno úplné uspořádání množiny s, kterou dostane na vstupu. Uspořádání provede podle Manhattanské vzdálenosti profilů z dané množiny od profilu $profile$. Funkce selectFirst(c, t) vrací prvních c prvků z úplného uspořádání t.

Funkce selectPeer() pracuje tak, jak bylo definováno v předchozí části:

```
function selectPeer(state.view)
    return random(selectFirst(c / 2, sort(state.view, state.profile)))
end
```

Funkce random(s) vrací jeden náhodný prvek z množiny s, kterou dostane na vstupu. Funkce selectPeer() tedy vrací jeden náhodný prvek z první poloviny jednoho z úplných uspořádání aktuálního pohledu.

4.2 Peer sampling

Množství aplikací Gossipu předpokládá, že mají k dispozici službu schopnou kterémukoli uzlu poskytnout adresu zcela náhodného jiného prvku vybraného z celé množiny zbývajících uzlů s rovnoměrným rozložením pravděpodobnosti. Tato služba se v anglicky psaných textech označuje jako *Peer Sampling* (rozhodl jsem se tento termín nepřekládat).

Služba Peer sampling poskytuje aplikacím funkci getPeer(), která vrací uzlu adresu jednoho prvku v ideálním případě s rovnoměrným rozložením pravděpodobnosti [17]. Tato funkce bývá typicky využívána funkcí selectPeer() dané aplikace. Pokud funkce selectPeer() vyžaduje takovýchto náhodných prvků více, zavolá funkci getPeer() vícekrát po sobě [17].

V rozsáhlých dynamických distribuovaných systémech je efektivní implementace služby Peer sampling velice obtížná. V těchto případech je totiž prakticky nemožné poskytnout každému uzlu *aktuální* seznam všech uzlů ostatních, aniž by to výrazně ovlivnilo výkonnost celého systému. Navíc, pokud by si každý uzel musel lokálně pamatovat seznam všech uzlů ostatních, výrazně by vzrostly požadavky na jeho paměťovou kapacitu.

Proto se objevila myšlenka implementovat samotnou tuto službu jako aplikaci Gossipu. Takovéto řešení je velice podobné gossipovému řešení pro vytváření a udržování topologie, popsanému v předchozí podkapitole. Největší rozdíl spočívá v tom, že zatímco u vytváření topologie konverguje překryvová síť do stabilního cílového stavu, zde je cíl zcela opačný – překryvová síť by měla být náhodná a co nejproměnlivější.

Existuje větší množství variant implementace této služby pomocí Gossipu. V práci *The Peer Sampling Service: Experimental Evaluation of Unstructured Gossip-Based Implementations* (kolektiv autorů: Márk Jelasity, et al.; [17]) je podrobně popsáno 27 různých variant, včetně popisu množství experimentů nad nimi provedených. Bohužel výsledky experimentů ukazují, že ani ty nejlepší z těchto variant nesplňují požadavky na dokonalou uniformitu rozložení navracených hodnot. Dá se však předpokládat, že pro mnohé aplikace jsou vlastnosti těchto řešení postačující [17].

Ve všech variantách implementace služby Peer sampling prostřednictvím Gossipu popsaných v článku je stav uzlu tvořen pohledem, což je (podobně jako u vytváření topologie) množina deskriptorů uzlů, jejíž maximální velikost je dána konstantou c (stejnou pro všechny uzly). Deskriptor uzlu se skládá z jeho adresy a z hodnoty *hop count* (rozhodl jsem se nepřekládat), což je celočíselná hodnota udávající dobu, po kterou je daná adresa přítomna v pohledech uzlů. Pohled by měl vždy obsahovat každou adresu maximálně jednou a neměl by obsahovat svou vlastní adresu – případ, kdy uzel sousedí sám se sebou, nedává smysl. Nad prvky pohledu existuje částečné uspořádání vzestupně podle hodnoty *hop count*.

Funkce update($state$, $state_{peer}$) provede nejprve sjednocení množin

$$state.view \cup state_{peer}.view \cup \{(peerAddress, 0)\}$$

kde $peerAddress$ je adresa prvku, od nějž získal uzel stavu $state_{peer}$; této adrese je přiřazena hodnota *hop count* 0 (je totiž do pohledu potenciálně vložena). Poté jsou z takto získané množiny deskriptorů odstraněny duplicitní adresy tak, že je ze skupiny deskriptorů se stejnými adresami ponechán vždy ten, který obsahuje nejnižší hodnotu *hop count*. Je potřeba také odstranit svou vlastní adresu, pokud se v této množině vyskytuje. Pokud je velikost výsledné množiny větší než c, vybere se z ní pro sestavení nového stavu právě c prvků (v opačném případě se samozřejmě použijí prvky všechny). Metoda výběru těchto c deskriptorů není přesně stanovena, experimenty popsané ve výše zmíněné práci ukazují, že je možné tyto prvky vybírat buď náhodně, nebo prvky seřadit vzestupně podle hodnoty *hop count* a použít prvních c prvků z této posloupnosti. Obě metody mají své výhody i nevýhody, žádná z nich však nedává zcela uspokojující výsledky.

Množina deskriptorů nemusí vždy obsahovat právě c prvků. V systému, kde uzly dynamicky vznikají a zanikají, může být problém inicializace prvku vyřešen tak, že nově vzniklý prvek bude znát zpočátku jen jeden další uzel – pak bude velikost jeho deskriptoru rovna jedné.

Funkce selectPeer() vybírá jeden prvek z aktuálního pohledu uzlu. Metoda výběru prvku z daného pohledu opět není přesně stanovena, nicméně z experimentů popsaných ve výše zmíněném článku vyplývá, že je vhodné používat buď náhodný prvek vybraný s rovnoměrným rozložením pravděpodobnosti, nebo vždy prvek s nejvyšší hodnotou *hop count*. Z hlediska praktické funkčnosti nejsou mezi těmito dvěma variantami velké rozdíly.

Použitá varianta Gossipu je *push-pull*. Z výsledků experimentů popsaných ve výše zmíněné práci vyplývá, že tato varianta má vždy výrazně lepší vlastnosti než varianty *push* nebo *pull*.

Funkce getPeer(), kterou aplikace Peer sampling poskytuje ostatním aplikacím, bude implementována tak, že vybere jeden náhodný prvek z aktuální množiny deskriptorů, a vrátí jeho adresu. Vzhledem k tomu, že funkce getPeer() může být volána vícekrát za sebou, je tento přístup výhodnější než výběr právě jednoho pevného (např. prvního nebo posledního) prvku z uspořádané množiny deskriptorů. Nicméně pokud cizí aplikace požaduje větší množství náhodných adres uzlů, měla by spíše požádat o podmnožinu aktuální množiny deskriptorů o určité velikosti.

Velkou výhodou této implementace služby Peer sampling je to, že vytváří proměnlivou plně propojenou náhodnou topologii i v případě systému s dynamicky vznikajícími a zanikajícími uzly. Protože každý uzel v systému neustále aktivně rozesílá některým ostatním uzlům svůj deskriptor s nulovou hodnotou *hop count* (a tedy s maximální možnou prioritou), nemůže se stát, že existující (tj. aktivní) uzel zůstane izolován. Nově vytvořený uzel se tak snadno zapojí do už existující sítě. Naopak uzel, který zanikl, bude po určité době zapomenut – přestanou se objevovat jeho deskriptory s nulovou hodnotou *hop count* a existující budou postupně vytlačeny. [21]

Dále je tato implementace relativně odolná vůči vzniku shluků. Jako shluk je zde chápána podmnožina uzlů, které jsou dobře propojeny vzájemně mezi sebou, ale existuje jen málo spojení mezi uzly z této podmnožiny a zbývajícími uzly. Chápejme nyní překryvovou síť vzniklou touto implementací v jednom konkrétním okamžiku jako *neorientovaný* graf $G = (V, H)$, kde množina uzlů V odpovídá množině uzlů systému a množina hrah H je definována jako $H = \{\{x, y\} | x \in V, y \in V,$ uzly x a y spolu mohou přímo komunikovat $\}$ (definice překryvové sítě jako neorientovaného grafu dává smysl, neboť v případě použité

varianty Gossipu *push-pull* proudí informace po hraně vždy oběma směry). Pak míru vytváření shluků můžeme vyjádřit pomocí shlukového koeficientu (*clustering coefficient*). Ten je definován jako průměr hodnot γ_x pro všechny uzly $x \in V$, kde

$$\gamma_x = \frac{|\{\{y,z\} \in H | y \in N_x, z \in N_x\}|}{|N_x|(|N_x|-1)}$$

kde N_x je množina sousedů uzlu x (tj. $N_x = \{y | \{x,y\} \in H\}$). Intuitivně můžeme shlukovací koeficient chápat jako průměrný počet všech hran mezi sousedy uzlu vydělený počtem všech možných hran mezi nimi. Příliš vysoký shlukovací koeficient negativně ovlivňuje šíření informací v síti [17]. Experimenty popsané ve výše zmíněném článku ukazují, že varianta s náhodným výběrem nového pohledu vede k menšímu shlukovacímu koeficientu, je tedy z tohoto pohledu výhodnější.

Mezi další zajímavé vlastnosti této implementace patří, že ve vzniklé dynamické překryvové síti bude relativně krátká průměrná vzdálenost mezi dvěma libovolnými uzly. Z experimentů popsaných ve výše zmíněném článku vyplývá, že varianta s náhodným výběrem nového pohledu vede k o něco menší průměrné vzdálenosti mezi uzly než varianta s výběrem prvních c prvků [17].

Z těchto závěrů by mohlo vyplývat, že varianta Peer samplingu s náhodným výběrem nového pohledu je výrazně lepší, než druhá metoda – sestavení nového pohledu z prvních c prvků posloupnosti. Není to zcela pravda. Varianta s náhodným výběrem nového pohledu má i své nevýhody: ve srovnání s druhou variantou je mnohem méně stabilní, vede k horší aproximaci náhodné topologie a má výrazně méně vyrovnanou distribuci stupňů uzlů (stupeň uzlu je počet sousedů daného uzlu ve výše definovaném neorientovaném grafu) [17]. Obecně se nedá jednoznačně říct, která z těchto dvou variant je lepší.

V literatuře je často zmiňován jako prostředek pro sestavování náhodné komunikační topologie protokol Newscast (např. [15], [16], [21]). Jedná se o variantu zde popsané služby Peer sampling, ve které je nový pohled sestavován z prvních c prvků posloupnosti a funkce selectPeer() vybírá komunikační protějšek náhodně [17].

4.2.1 Příklad implementace

Následuje příklad implementace konkrétní varianty aplikace Peer sampling. Jedná se o variantu, jejíž funkce update() vrací nižší část posloupnosti vzniklé sjednocením obou stavů. Funkce selectPeer() této varianty vrací náhodný prvek z aktuálního pohledu.

Stav uzlu je definován jako dvojice $(view, address)$, kde $view$ reprezentuje aktuální pohled daného uzlu a $address$ je jeho adresa. Pohled $view$ je seznam dvojic $(address, hopCount)$, kde $address$ je adresa daného uzlu a $hopCount$ je příslušná hodnota *hop count*. Tento seznam je uspořádán vzestupně podle hodnoty $hopCount$. Velikost pohledu je c.

Funkce update() je definována jako

function update($state$, $state_{peer}$)
    $updatedView$:= incrementHopCount($state.view$)
    $updatedPeersView$:= incrementHopCount($state_{peer}.view$)
    $mergedViews$:= merge($updatedView$, $updatedPeersView$, $[(state_{peer}.address, 0)]$)
    $newView$:= removeAddress(removeDuplicitAddr($mergedViews$), $state.address$)
    return (selectFirst(c, $newView$), $state.address$)
end

Funkce incrementHopCount(v) provede zvětšení všech hodnot *hop count* v daném pohledu v o hodnotu 1. Funkce merge(v_1, v_2, v_3) provede setřídění daných tří uspořádaných seznamů. Funkce removeDuplicitAddr(v) provede odstranění duplicitních adres z pohledu v tak, že ponechá vždy adresu s nejnižší hodnotou *hop count*. Funkce removeAddress(v, a) odstraní z daného pohledu v výskyty adresy a aktuálního prvku. Funkce selectFirst(c, v) slouží pro vybrání prvních c prvků z uspořádaného seznamu v.

Funkce selectPeer() je definována jako

```
function selectPeer(state)
    return lastAddress(state.view)
end
```

kde funkce lastAddress(v) vrací adresu z posledního prvku pohledu v.

4.3 Superpeerová topologie

Síťové modely, ve kterých jsou si všechny uzly rovny (*peer-to-peer* modely), popsané v předchozích částech, přinášejí výhodu velké odolnosti vůči chybám a selháním uzlů, avšak platí za to sníženou efektivitou v případě bezporuchového běhu. Uzly v takovýchto modelech pracují se značnou redundancí a provádějí množství nadbytečné komunikace. Oproti tomu standardní modely klient-server komunikují mnohem efektivněji, nicméně za cenu nízké odolnosti vůči poruchám některých uzlů (serverů).

Superpeerový model se snaží najít určitou „střední cestu“ mezi těmito dvěma extrémy – snaží se zkombinovat dobré vlastnosti obou modelů a odstranit jejich vlastnosti nežádoucí, tedy zkombinovat decentralizovanost a v důsledku toho odolnost vůči poruchám jednotlivých uzlů s efektivitou komunikace mezi hierarchicky uspořádanými uzly [21]. Tento model byl původně navržen pro potřeby aplikací pro sdílení souborů (*file sharing applications*), byl však úspěšně použit i pro další aplikace, např. v distribuovaných herních systémech [21].

V superpeerovém modelu je každému uzlu v systému přiřazena právě jedna ze dvou rolí: *superpeer* nebo *klient*. Superpeer je uzel, který na sebe vzal funkčnost serveru – obsluhuje několik klientů podobně jako server. Přiřazení těchto rolí uzlům není pevné, mechanismus aplikace může přimět uzel v roli klienta, aby se stal superpeerem, podobně superpeer může své výsady ztratit a stát se klientem.

Klienti komunikují pouze se superpeery, superpeery udržují komunikaci jak s klienty, tak i s dalšími superpeery. Existuje více variant superpeerové topologie – liší se typicky způsobem propojení superpeerů, počtem superpeerů, se kterými komunikuje jeden klient, a celkovým počtem superpeerů v závislosti na velikosti systému. Zde bude popsána varianta, ve které každý klient komunikuje pouze s jedním superpeerem, superpeery jsou vzájemně propojené náhodnou proměnlivou topologií podobnou té, kterou vytváří aplikace Peer sampling, a počet superpeerů v systému je minimální – aplikace se snaží najít takové uspořádání, aby počet superpeerů byl co možná nejmenší. V dynamickém systému, ve kterém se mohou objevit nové prvky a existující mohou zaniknout, je samozřejmě nemožné dosáhnout přesně topologie s těmito vlastnostmi, snahou v takovém případě tedy je tuto topologii aproximovat.

V této topologii tedy každý klient komunikuje právě s jedním uzlem, a to se superpeerem. Každý superpeer komunikuje jednak s množinou klientů, které jsou mu přiřazeny, jednak s náhodným a proměnlivým vzorkem dalších superpeerů. V rámci propojení superpeerů neexistuje žádný centrální prvek – všechny superpeery jsou si rovny.

Při praktické implementaci takovéhoto modelu, např. v aplikaci pro sdílení souborů, se typicky superpeery stávají uzly, které jsou výkonnější, dostupnější a kvalitněji připojené ke zbytku sítě, než uzly ostatní. „Obyčejné“ uzly zůstávají klienty. Hlavním úkolem superpeerů je zajišťovat komunikaci mezi ostatními uzly (bez ohledu na to, jestli se jedná o superpeery nebo o klienty). Superpeery také přebírají část odpovědnosti za klienty, které jsou jim přiřazeny – např. v aplikaci pro sdílení souborů si superpeery udržují informace o souborech sdílených klienty, místo svých klientů se pak podílejí na protokolech pro vyhledávání souborů, čímž výrazně snižují celkový provoz na síti [21].

Vytváření a udržování takovéto topologie je zjevně dosti náročný úkol, a to zvláště v dynamickém prostředí, kde za běhu systému vznikají nové uzly a zanikají uzly existující. Mechanismus spravující síť musí rozhodnout, které uzly se stanou superpeery a které klienty. Vzhledem k povaze dané topologie se nabízí ji sestavovat prostřednictvím gossipového protokolu. Takovýto protokol byl popsán A. Montresorem v práci *A Robust Protocol for Building Superpeer Overlay Topologies* ([21]). Zde popsaný protokol se od protokolu popsaného v dané práci částečně odlišuje – domnívám se, že tamní varianta je zbytečně komplikovaná.

Předpokládejme, že každému uzlu je přiřazena jeho *kapacita* c, která vyjadřuje maximální počet klientů, o které by se daný prvek zvládl postarat, kdyby měl roli superpeeru. Kapacita tedy souhrnně vyjadřuje vlastnosti uzlu jako jeho výpočetní výkon, kvalitu jeho připojení ke zbytku sítě, atp. Předpokládejme, že každý uzel svou vlastní kapacitu zná.

Ačkoli se jedná o aplikaci vytvářející překryvovou síť, sama využívá hned tří nižších vrstev vytvářejících své vlastní překryvové sítě. Termín „nižší vrstvy“ zde nemusí být zcela nejvhodnější, protože tyto vrstvy využívají informací získaných z vrstvy nejvyšší – tedy z vlastní aplikace pro sestavování superpeerové topologie. Propojení mezi vrstvami je tedy obousměrné.

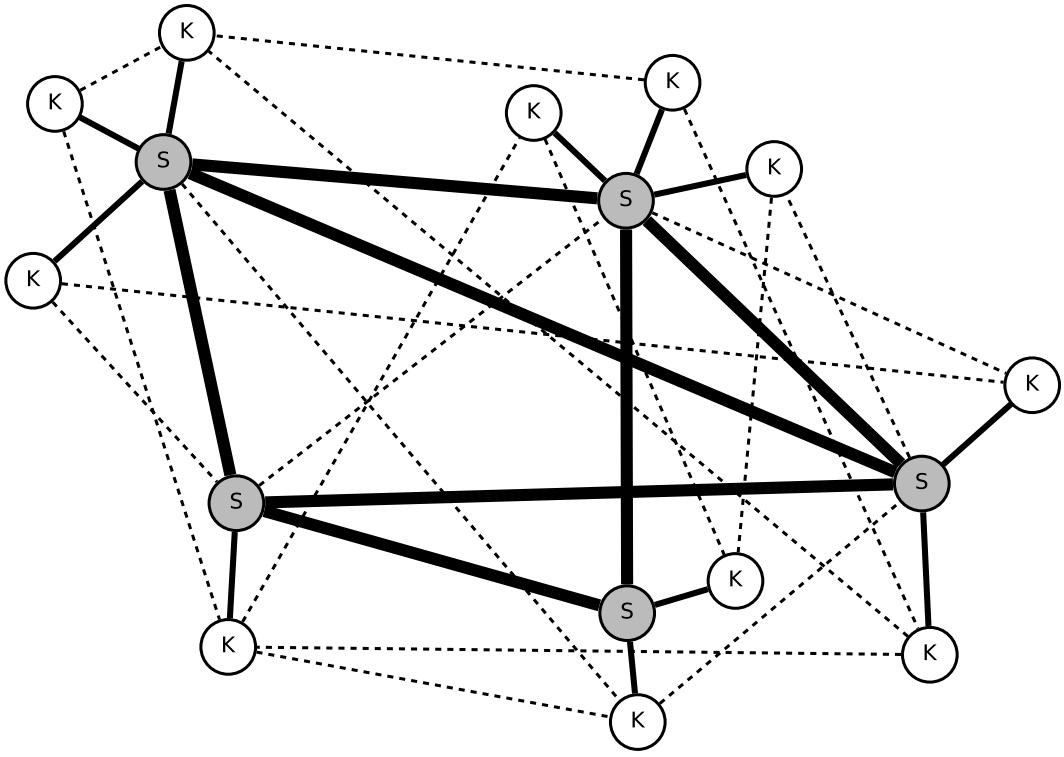
Zcela nejnižší vrstvu tvoří aplikace *Peer sampling*. Je možné využít libovolnou variantu aplikace Peer sampling popsané v části 4.2. Úkolem této vrstvy je udržovat skrytou plně propojenou síť, která umožní opravy překryvových sítí vyšších vrstev v případě poruch některých uzlů – zamezí rozpadu sítě na několik vzájemně nepropojených částí. Její chování odpovídá v postatě přesně chování některé varianty standardní aplikace Peer sampling s tím rozdílem, že součástí deskriptoru uzlu jsou vedle jeho adresy a hodnoty hop count navíc také informace o jeho *kapacitě* a *roli* v době vzniku deskriptoru, součástí informace o roli superpeeru by měla být i informace o velikosti množiny jemu přiřazených klientů (aby bylo možno odvodit, jestli je daný superpeer plně využit). Tyto doplňkové informace tato aplikace přímo nijak nevyužívá, jsou důležité pouze jako informace poskytovaná vyšším vrstvám. Vzhledem k tomu, že by síť měla konvergovat do stabilního stavu, můžeme předpokládat, že ve většině případů odpovídá role uvedená v deskriptoru aktuální roli odpovídajícího uzlu – pravděpodobnost, že informace v deskriptoru již neodpovídá realitě je velmi nízká.

Střední vrstvu tvoří dvě velice podobné aplikace: *Underloaded sampling* a *Superpeer sampling*. Aplikace *Superpeer sampling* poskytuje uzlům naší aplikace náhodný vzorek superpeerů v systému – slouží tak pro udržování náhodné topologie mezi superpeery. Aplikace *Underloaded sampling* poskytuje informace pouze o těch superpeerech, které nejsou zcela využity – tedy o takových superpeerech, jejichž počet jim přiřazených klientů je nižší než jejich kapacita.

Obě tyto aplikace pracují podobně jako aplikace Peer sampling. Stav obou aplikací je tvořen pohledem a deskriptorem daného uzlu. Hlavním rozdílem oproti standardnímu chování aplikace Peer sampling je to, že funkce selectPeer() nevybírá náhodný prvek z pohledu

(tedy *view*) své vlastní aplikace, ale z pohledu aplikace *Peer sampling*, tedy nejnižší vrstvy. To umožňuje udržování plně propojených sítí pro tyto aplikace i v případě hromadného selhání superpeerů. Informace o superpeerech je šířena mezi všemi uzly systému, tedy nikoli pouze mezi uzly náležejícími do dané kategorie.

Druhým rozdílem je, že metoda update() provede přidání deskriptoru komunikačního partnera do nového pohledu jen tehdy, odpovídá-li jeho role dané aplikaci. Tedy v případě aplikace *Superpeer sampling* dochází ke vložení deskriptoru komunikačního partnera jen tehdy, když má tento komunikační partner roli superpeeru. V případě *Underloaded sampling* k tomuto vložení dojde jen tehdy, je-li komunikační partner v roli superpeeru, jehož kapacita není zcela využita.



Obrázek 4.1: Příklad superpeerové topologie. Uzly v roli klienta jsou označeny písmenem K, uzly v roli superpeeru jsou označeny podbarvením a písmenem S. Nejtlustší hrany reprezentují náhodné propojení mezi superpeery, to je vytvářeno aplikací *Superpeer sampling*. Středně tlusté hrany reprezentují propojení mezi superpeery a klienty, to je vytvářeno aplikací *Superpeer topology*. Tenké čárkované hrany reprezentují náhodné propojení mezi uzly (tam, kde není překryto hranou tlustší) vytvářené aplikací *Peer sampling*. Aplikace *Underloaded sampling* slouží k distribuci informací o superpeerech, které mají nenaplněnou kapacitu.

Nejvyšší vrstva je tvořena samotnou aplikací pro vytváření a udržování superpeerové topologie – aplikace *Superpeer topology*. Chování této aplikace nelze zcela přesně vložit do kostry gossipové aplikace popsané v části 3.1.

Stav uzlu je závislý na jeho aktuální roli. Součástí stavu *klienta* je informace o jeho superpeeru: jeho adresa a doba, která uplynula od poslední zprávy od něj obdržené. Tento superpeer je typicky jediný uzel, se kterým za běžných okolností klient (v rámci této apli-

kace) komunikuje. Aktivní vlákno klienta provádí pravidelné dotazování svého superpeeru, jestli je v pořádku.

Pokud dojde u superpeeru k poruše, což se projeví například tím, že superpeer už dlouho svému klientovi nedal najevo, že je v pořádku, musí být klient schopen si najít nový superpeer – pokusí se připojit k jednomu náhodnému prvku z množiny poskytované aplikací *Underloaded sampling*. Pokud není ani tato aplikace schopna klientu poskytnout nový superpeer, stává se klient sám novým superpeerem s prázdnou množinou klientů.

Stav superpeeru je tvořen množinou deskriptorů mu přidělených klientů. Deskriptor klienta se skládá z jeho adresy, informace o jeho kapacitě a informace o tom, před jakou dobou obdržel superpeer od daného klienta naposledy zprávu. Pokud se klient svému superpeeru příliš dlouho neozval, značí to jeho poruchu a klient je z množiny deskriptorů odstraněn.

Aktivní vlákno superpeeru se snaží přesouvat své klienty těm superpeerům z množiny poskytované aplikací *Underloaded sampling*, které mají větší kapacitu než je kapacita tohoto uzlu. Aktivní vlákno superpeeru vybere náhodný kapacitně nevyužitý superpeer s kapacitou větší než je jeho vlastní kapacita a zašle mu celý svůj stav.

Superpeer, který obdrží takovýto stav, z něj musí převzít co největší množství klientů a zaslat odesilateli toho stavu informaci o takto odebraných klientech – ten si je musí odstranit ze svého stavu. Takto dochází k přesunu klientů k superpeerům s větší kapacitou. Pokud se takto stane, že určitý superpeer s malou kapacitou zůstane zcela bez klientů, stává se sám klientem. Pokud superpeer, který tímto mechanismem získal nové klienty, zjistí, že mezi jeho klienty je jeden, jehož kapacita je větší než kapacita jeho komunikačního partnera, přinutí svého komunikačního partnera, aby se stal klientem (bez ohledu na to, jestli mu zbyli nějací klienti nebo ne) a toho nalezeného klienta s velkou kapacitou přiměje stát se superpeerem a převzít zbylé klienty od daného právě zaniklého superpeeru.

Pseudokódem zapsané chování superpeeru při obdržení stavu jiného superpeeru:

```
1:  on receive(peer)
2:        receive state_peer from peer
3:        newClients := chooseNewClients(state_peer, myCapacity−size(state))
4:        client := maxCapacity(state ∪ newClients)
5:        if (client.capacity > peer.capacity)
5:              removeAndMoveYourSuperpeerStatus(peer, newClients, client)
6:        else
7:              remove(peer, newClients)
8:        end
9:        newSuperpeer(newClients)
10:       state := state ∪ newClients
11:end
```

Funkce chooseNewClients(*set*, *n*) se chová takto: pokud je velikost množiny *set* větší než *n*, vrací právě *n* prvků vybraných (náhodně) z množiny *set*, jinak vrací celou množinu. Zde tato funkce slouží pro výběr těch prvků, které budou odebrány ze stavu komunikačního partnera $state_{peer}$.

Funkce maxCapacity(*set*) vrací z dané množiny deskriptorů klientů *set* klienta s největší kapacitou *client*. Kapacita klienta získaného touto funkcí se poté porovná s kapacitou komunikačního partnera tohoto uzlu. Pokud je kapacita klienta *client* větší než kapacita komunikačního partnera, provede se funkce removeAndMoveYourSuperpeerStatus(*peer*, *set*, *client*), která pošle komunikačnímu partneru *peer* příkaz (ve formě zaslání speciální zprávy), který jej přiměje odstranit množinu *set* ze svého stavu, tento upravený stav zaslat klientu

client s příkazem, aby se stal superpeerem, a sám se stát klientem. V důsledku provedení tohoto příkazu se tedy *peer* stane klientem a *client* stane superpeerem, který přebral zbytek klientů uzlu *peer*.

Funkce remove(*sp*, *set*) pošle superpeeru *sp* příkaz, aby odebral ze svého stavu (tedy množiny deskriptorů) všechny prvky množiny *set*. Pokud *sp* zjistí, že byly odebrány všechny prvky jeho stavu, stane se klientem.

Funkce newSuperpeer(*set*) uvědomí všechny klienty z množiny *set*, že se daný uzel stal jejich novým superpeerem.

4.3.1 Příklad implementace

Následuje popis konkrétního příkladu implementace aplikace *Superpeer topology*. Vedle této aplikace musejí být přítomny ještě tři další, pojmenované *Superpeer sampling*, *Underloaded sampling* a *Peer sampling*. Všechny tyto tři aplikace již byly dostatečně popsány, navíc jsou triviálně odvozeny od obecné aplikace Peer sampling popsané v části 4.2.

Stav aplikace pro sestavení superpeerové topologie obsahuje informaci, jestli je daný uzel klient nebo superpeer, doplněnou o další údaje v závislosti na této informaci. V případě klienta je stav tvořen čtveřicí $(client, Superpeer, LastCheck, Capacity)$, kde $Superpeer$ je adresa superpeeru přiřazeného tomuto klientovi, $LastCheck$ je informace o počtu cyklů provedených od posledního kontaktu s tímto superpeerem a $Capacity$ je kapacita tohoto uzlu, kdyby se stal superpeerem.

Stav superpeeru je tvořen trojicí $(superpeer, ClientSet, Capacity)$, kde $Capacity$ je kapacita tohoto uzlu a $ClientSet$ je množina deskriptorů klientů o velikosti menší nebo rovné kapacitě tohoto uzlu. Deskriptor klienta je tvořen trojicí $(Address, LastCheck, Capacity)$, kde $Address$ a $Capacity$ jsou adresa a kapacita daného klienta a $LastCheck$ je počet cyklů provedených od posledního kontaktu s daným klientem.

Hodnoty *superpeer* a *client* jsou pouhé konstanty určující aktuální roli daného uzlu. Hodnotu $Capacity$ klienta i superpeeru můžeme považovat za neměnnou po celou dobu běhu systému. Tato hodnota by měla být dána fyzickými vlastnostmi daného uzlu – jeho schopností spravovávat další uzly.

Hodnota $LastCheck$ klienta je na konci každého cyklu inkrementována a při každém obdržení zprávy od superpeeru vynulována. Pokud tato hodnota přesáhne jistou akceptovatelnou hranici, začne klient považovat svůj superpeer za nedostupný a vyhledá superpeer nový. Obdobně se chovají také hodnoty $LastCheck$ v deskriptorech obsažených v množině $ClientSet$ superpeeru – jsou vynulovány při každém kontaktu s daným klientem, jsou inkrementovány po provedení každého cyklu a v případě, kdy hodnota $LastCheck$ přesáhne určitou akceptovatelnou hranici, bude daný deskriptor z množiny $ClientSet$ vyřazen.

Protože je v rámci aplikace zasíláno množství zpráv s různým významem, jsou jednotlivým typům zpráv přiřazena jména. Jméno zprávy je zvoleno podle její sémantiky. Pro kontrolu vztahu mezi klientem a superpeerem se používají zprávy *areYouMySuperpeer* a *iAmYourSuperpeer*. Pro mechanismus výměny klientů mezi superpeery se používají zprávy *chooseClients*, *becomeClient*, *becomeSuperpeer* a *removeClients*. Klient se pokouší přiřadit se k novému superpeeru pomocí zprávy *becomeMySuperpeer*. Zpráva *notYourSuperpeer* slouží pro upozorňování na chybné informace o superpeeru u klienta.

Činnost jednoho cyklu aktivního vlákna klienta spočívá pouze v zaslání zprávy *areYouMySuperpeer* svému superpeeru. Zpráva *areYouMySuperpeer* musí obsahovat adresu tohoto klienta.

V jednomu cyklu aktivního vlákna superpeeru se provede výběr takového uzlu z množiny

poskytované aplikací *Underloaded sampling*, který má větší kapacitu než daný superpeer, a tomuto uzlu se pošle zpráva *chooseClients* doplněná o množinu klientů tohoto superpeeru a o kapacitu tohoto superpeeru.

Ačkoli pasivní vlákno provádí v podstatě pouze reakce na obdržené zprávy, je jeho chování ve srovnání s chováním aktivního vlákna podstatně komplikovanější.

Pokud superpeer obdrží zprávu *chooseClients* obsahující neprázdnou množinu deskriptorů klientů *Clients*, vybere z této množiny podmnožinu *newClients* tak velkou, aby ji mohl doplnit do své množiny deskriptorů. Poté se pokusí v celé množině *Clients* nalézt klienta, jehož kapacita je větší, než kapacita superpeeru, který tomuto uzlu tuto zprávu poslal. Pokud takového klienta najde, zašle danému superpeeru zprávu *becomeClient* doplněnou o adresu toho nalezeného klienta a o množinu odebraných klientů *newClients*. V opačném případě, tedy pokud v množině *Clients* takový prvek není, zašle danému superpeeru zprávu *removeClients* doplněnou o množinu *newClients*.

Pokud superpeer obdrží zprávu *becomeClient* obsahující adresu klienta *Cl*, jenž se má stát superpeerem, a množinu deskriptorů klientů *newClients*, odebere ze své množiny *ClientSet* danou množinu *newClients* a zašle zbytek množiny *ClientSet* danému klientu *Cl* ve zprávě *becomeSuperpeer*.

Pokud superpeer obdrží zprávu *removeClients* obsahující množinu deskriptorů, odebere tuto množinu ze své množiny *ClientSet*. Pokud po odebrání daných deskriptorů bude množina *ClientSet* prázdná, stává se tento uzel klientem – vybere z množiny poskytované aplikací *Underloaded sampling* adresu vhodného superpeeru a zašle mu zprávu *becomeMySuperpeer* doplněnou o svou adresu a kapacitu.

Reakce superpeeru na obdržení zprávy *becomeMySuperpeer* (zpráva obsahuje adresu a kapacitu klienta *Cl*): pokud je velikost množiny *ClientSet* menší než kapacita tohoto uzlu, bude do této množiny přidán deskriptor klienta *Cl* a tomuto klientu bude zaslána zpráva *iAmYourSuperpeer*. V opačném případě mu bude zaslána zpráva *notYourSuperpeer*.

Pokud superpeer obdrží zprávu *areYouMySuperpeer* od klienta, jehož deskriptor je přítomen v množině *ClientSet*, zašle tomuto klientu zprávu *iAmYourSuperpeer* doplněnou o svou adresu a nastaví hodnotu *LastCheck* v deskriptoru tohoto klienta na nulu. Pokud deskriptor daného klienta není v množině *ClientSet* přítomen, zašle mu zprávu *notYourSuperpeer*.

Chování pasivního vlákna klienta je následující. Pokud obdrží zprávu *iAmYourSuperpeer* s adresou superpeeru, začne tuto adresu považovat za adresu svého superpeeru a nastaví hodnotu *LastCheck* na nulu.

Pokud klient obdrží zprávu *becomeSuperpeer* obsahující množinu deskriptorů, stane se superpeerem, danou množinu deskriptorů dosadí do *ClientSet* a všem uzlům z této množiny zašle zprávu *iAmYourSuperpeer*.

4.4 Šíření informací

Šíření informací (*information dissemination*) mezi uzly je patrně nejpřirozenější aplikací protokolu Gossip, ten totiž byl původně za tímto účelem navržen. Jak naznačuje název protokolu (*gossip* – šířit klepy, pomlouvat), Gossip je inspirován překvapující rychlostí a efektivitou, jakou se v reálném světě šíří klepy a pomluvy. Na základě čistě lokální komunikace mezi účastníky tohoto procesu dochází mezi nimi k překvapivě spolehlivému rozšíření dané informace. Každý zúčastněný přitom jedná sám za sebe, bez nějakého vnějšího řízení, komunikuje pouze s omezeným počtem lidí ve svém okolí a sám se rozhoduje, co komu řekne [15].

Ačkoli je aplikace protokolu Gossip pro šíření informací odvozená od výše popsaného mechanismu šíření klepů, má šíření informací prostřednictvím Gossipu několik odlišností. Při šíření klepů se účastník tohoto procesu baví typicky jen s přáteli, známými a s rodinnými příslušníky – tedy jen s relativně omezeným okruhem osob. V gossipovém protokolu uzel komunikuje s uzlem náhodně vybraným z celé zbývající populace, může tedy komunikovat s naprosto libovolným uzlem. Dále se v reálném šíření klepů lidé rozhodují, jaké informace dotyčnému komunikačnímu partneru sdělí, mohou mu tedy nějakou informaci zatajit, případně mohou s danou šířenou informací nějakým způsobem manipulovat. Toto by bylo pro účely spolehlivého šíření informací nepřípustné – v aplikaci Gossipu dochází ke vzájemné výměně všech informací mezi oběma komunikujícími prvky [15].

Problém řešený touto aplikací vypadá následovně: Mějme rozsáhlý distribuovaný systém tvořený množinou uzlů. Všechny uzly jsou si rovny ve smyslu, že neexistuje žádný centrální řídící uzel či uzel nějakým obdobným způsobem privilegovaný. V tomto systému jsou na jednotlivých uzlech umístěny repliky databáze. Databáze je tvořena množinou klíčů, kterým jsou přiřazeny hodnoty a časové známky jejich posledních úprav. Uzel může provádět lokální úpravy obsahu databáze – může přidávat, odebírat a měnit hodnoty v databázi. Našim cílem je, aby měly všechny databáze na jednotlivých uzlech stejný obsah, tedy pokud dojde ke změně, aby se tato změna rozšířila mezi zbývající uzly databáze – jedná se tedy o udržování replikovaných databází.

Aplikace Gossipu pro řešení tohoto problému vypadá takto [5]: Používá se varianta *push-pull*, stav každého uzlu je dán aktuálním stavem jeho databáze. Metoda update() provede porovnání obou verzí databáze (tj. místní a získané od komunikačního partnera) a vrátí novou verzi danou složením daných dvou a vyřešením konfliktů mezi nimi. Výběr komunikačního partnera je náhodný nad celou množinou uzlů s rovnoměrným rozložením.

Výhodou tohoto přístupu je značná robustnost – ztráta zprávy nebo selhání či dokonce zánik jednoho uzlu nebudou mít výrazný vliv na funkčnost celku [15][5]. Další výhodou je značná jednoduchost popsaného způsobu komunikace a jeho rychlost – optimální varianta provede rozšíření jedné změny databáze po celém systému s vysokou pravděpodobností v čase $O(\log n)$, kde n je počet uzlů v síti [15][5].

4.4.1 Příklad implementace

Předpokládejme nejprve, že všechny uzly znají aktuální čas. Stav uzlu *state* je tvořen množinou databázových záznamů představovaných trojicemi $(key, value, timestamp)$. Hodnota key značí klíč daného záznamu, $value$ jeho hodnotu a $timestamp$ představuje časovou známku poslední změny daného záznamu. Smazání záznamu je řešeno pomocí přepsání hodnoty $value$ na nějakou speciální hodnotu značící smazaný záznam.

Funkce update() provede následující akci

function update($state$, $state_{peer}$)
    return resolveDB($state \cup state_{peer}$)
end

Funkce resolveDB() upraví množinu získanou sjednocením obou stavů tak, aby v ní byla každému klíči přiřazena maximálně jedna hodnota a to vždy ta nejnovější. Tj. nalezne-li v množině několik trojic se stejným klíčem key, ponechá pouze tu s nejnovější časovou známkou.

Funkce selectPeer() vrací náhodný prvek vybraný z celé množiny zbývajících uzlů s rovnoměrným rozložením pravděpodobnosti. Použita je varianta Gossipu *push-pull*.

Předpoklad, že všechny uzly znají aktuální čas, je problematický – Gossip nám nic takového nenabízí. Nicméně aktuální čas je potřeba v podstatě jen pro vytváření časových známek při úpravě databáze (slouží jako prostředek pro vyjádření preferencí nad záznamy se stejným klíčem). K úpravě databáze dochází zásahem z vnějšku. Můžeme tedy předpokládat, že se uzel aktuální čas dozví od toho vnějšího zdroje, který danou změnu databáze vyvolal.

Další možností v případě fyzické realizace je, že každý uzel bude mít přístup ke svým vlastním hodinám udávajícím aktuální čas. V tom případě se problém redukuje na pouhou synchronizaci hodin jednotlivých uzlů, aby všechny ukazovaly stejný čas (který nemusí mít nic společného s časem reálným). Předpokládejme, že doba výpočtu jednotlivých funkcí Gossipu je natolik malá, že ji s ohledem na požadovanou přesnost synchronizace hodin můžeme zanedbat. Pak lze hodiny snadno synchronizovat pomocí aplikace Gossipu pro výpočet průměrné hodnoty s tím, že stav uzlu nahradíme aktuálním lokálním časem daného uzlu v době volání funkce update(). Tento postup zajistí, že rozdíly mezi hodinami jednotlivých uzlů budou konvergovat k nule. Výpočet aritmetického průměru je popsán v následující podkapitole.

4.5 Pravděpodobnostní multicast

S využitím aplikací zajišťujících šíření informací je možné realizovat řadu praktických aplikací. Vedle výše popsaného udržování replikovaných databází je to i například šíření updatů softwaru, multicast na aplikační úrovni, atp.

Vedle implementace šíření informací popsané v předchozí podkapitole existují i další způsoby realizace této aplikace. Zatímco aplikace z předchozí podkapitoly realizuje udržování stále se měnící replikované databáze, zde popíšu metodu vhodnou spíše pro realizaci pravděpodobnostního multicastu [18]. Samozřejmě by bylo možné realizovat multicast i s využitím aplikace předchozí, ale tato aplikace se zdá být pro daný účel vhodnější.

Pseudokód této aplikace není možné zapsat jednoduše ve tvaru obecné kostry Gossipu uvedené v kapitole 3.1. Neexistuje zde rozdělení na aktivní a pasivní vlákno – existuje pouze pasivní chování, kdy aplikace reaguje na příchozí zprávu. Pseudokód aplikace [18]:

```
1: on receive(peer)
2:         receive message from peer
3:         if message ∉ state
4:                 state := update(state, message)
5:                 peerSet := selectPeerSet(k)
6:                 send message to peerSet
7:         end
8: end
```

Stav uzlu (*state*) je dán množinou již přijatých multicastových zpráv. Funkce update(*state*, *message*) provádí přidání nově přijaté zprávy do množiny přijatých zpráv. Je vhodné stanovit určitou maximální velikost této množiny a v případě jejího překročení z ní nejstarší zprávy (tj. ty pravděpodobně již irelevantní) odstraňovat.

Funkce selectPeerSet(k) vrací množinu adres uzlů o velikosti k. Předpokládejme, že aplikace má k dispozici službu Peer sampling (tak, jak byla popsána v části 4.2) a že má k dispozici celý její pohled (tj. celou množinu adres uzlů, kterou si služba Peer sampling udržuje). Tento pohled má právě l prvků a $k \leq l$. Funkce selectPeerSet(k) vrací náhodnou k-prvkovou podmnožinu tohoto aktuálního pohledu.

Veškeré chování uzlu se děje jako reakce na příchozí zprávu. Uzel nejprve zkontroluje, jestli danou zprávu již neobdržel. Každý uzel by měl v ideálním případě reagovat na každou zprávu právě jednou – další obdržení stejné zprávy by měl ignorovat. Při obdržení nové zprávy vybere uzel pomocí funkce selectPeerSet() náhodně množinu k adres jiných uzlů a všem těmto uzlům přepošle nově získanou zprávu. Informace se šíří pouze ve směru od uzlu navazujícího komunikaci k jeho komunikačnímu partneru, jedná se tedy o obdobu varianty *push*.

Podstatnou vlastností aplikace je, že v důsledku neexistence aktivního vlákna uzly nekomunikují neustále, ale pouze v průběhu šíření nové zprávy – tj. pouze tehdy, když je komunikace potřeba.

Zpráva, kterou uzel obdrží, může pocházet ze dvou různých zdrojů. Jednak ji může získat od jiného uzlu ze systému (jedná se tedy o komunikaci v průběhu šíření zprávy systémem), nebo ji může obdržet z vnějšího prostředí – při inicializaci rozeslání zprávy.

Samozřejmým cílem multicastu je, aby zpráva rozeslaná jediným uzlem byla rozšířena s co nejvyšší pravděpodobností mezi všechny ostatní existující (provozuschopné) uzly systému a to i přes možné poruchy, ztráty zpráv, atp.

Mějme pro každou multicastem šířenou zprávu jeden orientovaný graf, jehož množina uzlů je totožná s množinou uzlů systému a množina hran je určená množinami vracenými funkcí selectPeerSet(k) tak, že hrana (x, y) je přítomna právě tehdy, pokud je uzel y přítomen v dané množině komunikačních partnerů uzlu x (tj. v množině vrácené funkcí selectPeerSet(k) volané kódem uzlu x). Nechť systém má n uzlů a každý uzel x má k_x sousedů. Z analýzy popsané v článku *Probabilistic Reliable Dissemination in Large-Scale Systems* ([18]) vyplývá, že pokud je průměrná velikost k_x pro všechny uzly x (tj. průměrný počet sousedů každého uzlu) větší než $\ln n$, je vysoká pravděpodobnost, že graf je propojený a tedy že zpráva rozeslaná z jednoho uzlu bude doručena uzlům všem.

4.6 Agregace dat

Další možnou aplikací Gossipu je získávání globální informace z lokálních dat, konkrétně počítání nějaké globální agregační funkce nad daty přítomnými lokálně, tj. v jednotlivých uzlech. Příkladem využití takovéto aplikace může být určení celkové paměťové kapacity v distribuovaném počítačovém systému, nejrůznější průměry či extrémy parametrů jednotlivých počítačů tvořících uzly, atd. Pro jednoduchost předpokládáme, že data, ze kterých danou agregační funkci počítáme, jsou rozmístěna tak, že každý uzel v systému má právě jednu datovou položku (vstupních hodnot je tedy stejný počet jako uzlů).

Prostřednictvím této aplikace Gossipu lze určit globální hodnoty jako jsou aritmetický či geometrický průměr daných lokálních hodnot či jejich extrémy (globální minimum a maximum). Jiné hodnoty, jako jsou sumy či standardní deviace daných dat se dají spočítat, známe-li počet prvků v systému, pomocí aritmetických výrazů nad získanými hodnotami [5].

Jeden z možných přístupů k výpočtu aritmetického průměru [5]: Nechť stav uzlu je aktuální lokální aproximace skutečného průměru. Počáteční stav bude mít hodnotu datové položky přiřazené danému uzlu. Funkce update() nastavuje jako nový stav průměr z původní hodnoty stavu uzlu a hodnoty stavu komunikačního partnera.

Funkce selectPeer() vrací náhodně vybraný uzel s rovnoměrným rozložením pravděpodobnosti. Použitá varianta Gossipu je *push-pull* – u této varianty dochází k volání funkce update() u obou komunikujících uzlů, tudíž po ukončení komunikace budou oběma uzlům

nastaveny stejné hodnoty stavu bez ohledu na to, který z nich danou komunikaci inicioval. Takovýto způsob komunikace zachovává konstantní součet hodnot v systému [5].

Je zřejmé, že díky tomu, že se po každé komunikaci zprůměrují dvě hodnoty a zároveň se nezmění celková suma všech hodnot v systému, konverguje celý systém k celkovému průměru všech hodnot – na konci bude mít každý uzel k dispozici výslednou hodnotu. Nicméně rychlost konvergence je exponenciální [5].

Obdobný postup je možné využít i pro výpočet jiných hodnot něž aritmetického průměru, například pro určení minima či maxima hodnot v systému.

4.6.1 Příklad implementace aritmetického průměru

Označme množinu uzlů systému jako V. Nechť je každému uzlu $i \in V$ přiřazena číselná hodnota r_i. Úkolem je spočítat aritmetický průměr hodnot r_i pro všechny uzly i, tj. hodnotu $\frac{\sum_{i \in V} r_i}{|V|}$.

Stav uzlu i, $state_i$, je definován jako reálné číslo – aktuální lokální aproximace výsledného průměru. Na počátku, tj. před spuštěním algoritmu, platí pro každý uzel, že $state_i = r_i$.

Funkce update() je definována jako

function update($state$, $state_{peer}$)
    return ($state$ + $state_{peer}$) / 2
end

Funkce selectPeer() vrací uzlu i adresu uzlu vybraného z množiny $V - \{i\}$ náhodně s rovnoměrným rozložením pravděpodobnosti. Stav uzlu $state_i$ není brán v úvahu.

Je použita varianta Gossipu *push-pull*.

4.6.2 Příklad implementace minimální hodnoty

Podobně jako v minulém případě: Každému uzlu i je přiřazena hodnota r_i. Tentokrát je cílem určit minimum ze všech hodnot r_i, tj. hodnotu $\min_{i \in V} r_i$.

Stav uzlu je definován obdobně jako v předchozím případě, tj. jako reálné číslo reprezentující aktuální lokální aproximaci hledaného minima.

Funkce update():

function update($state$, $state_{peer}$)
    return min($state$, $state_{peer}$)
end

Funkce selectPeer() je definována stejně jako v minulém případě, tj. vrací adresu náhodně vybraného uzlu s rovnoměrným rozložením pravděpodobnosti. Opět je použita varianta *push-pull*.

Kapitola 5

Framework pro modelování a simulaci gossipových systémů

Jedním z cílů této diplomové práce je návrh a implementace frameworku sloužícího pro modelování a simulaci gossipových systémů.

5.1 Analýza a návrh

Cílem je vytvořit systém, který by uživateli usnadňoval vytváření modelů aplikací Gossipu a dovoloval mu s těmito modely provádět simulační experimenty.

Gossipové systémy se skládají z velkého množství paralelně běžících uzlů, z nichž každý provádí opakovaně typicky relativně jednoduchý kód, který je pro všechny uzly stejný. Podstatnou součástí funkce uzlu je komunikace s nějakým jiným uzlem.

Pro specifikaci aplikace Gossipu je podstatné zadat kód, který budou jednotlivé uzly vykonávat. Vytvoření modelu gossipového systému by tedy mělo spočívat v zadání kódu uzlů v nějaké vhodné podobě a ve stanovení počtu uzlů v systému.

Protože pro většinu aplikací Gossipu platí, že chování jejich uzlů kopíruje schéma popsané v části 3.1, nabízí se tohoto jednoduchého schématu využít pro zadávání popisu systému. V takovémto případě uživatel explicitně zadává následující položky:

- deklarace datového typu lokálního stavu uzlu
- definice funkce update()
- definice funkce selectPeer()
- varianta Gossipu (*push*, *pull* nebo *push-pull*)
- definice funkce wait()
- inicializace stavu uzlů

Vedle toho musí uživatel též určovat počet uzlů v systému, ten však nemusí být v průběhu simulace konstantní.

Tento způsob zadávání aplikace Gossipu je sice velice jednoduchý, avšak je vhodný pouze pro zadávání chování *typického* gossipového systému kopírujícího přesně danou kostru. Vedle takovýchto aplikací Gossipu však existují i aplikace, které tomuto omezenému schématu

zcela přesně neodpovídají, ačkoli k němu mají určitým způsobem blízko (vizte např. část 4.5). Framework by měl podporovat i modelování takovýchto méně typických systémů.

Framework by tedy měl uživateli umožnit zadávat model vedle výše popsané varianty i podstatně obecnějším způsobem. Vhodná se zdá být možnost explicitně zadat chování aktivního a pasivního vlákna uzlu. Dá se předpokládat, že v takovém případě může být zadávání modelu o něco náročnější. Nemyslím však, že by ve většině případů byl vzrůst náročnosti zadávání modelu nějak nepřijatelně velký – dá se předpokládat, že ve většině případů bude popis obou vláken svou komplikovaností odpovídat složitosti pseudokódů chování aktivního i pasivního vlákna typické aplikace Gossipu z části 3.1, které jsou velice jednoduché.

Gossipové systémy s komplikovanějším chováním se běžně skládají z několika komponent, z nichž každá představuje samostatnou aplikaci. Framework by tedy měl umožňovat specifikovat systém pomocí množiny vzájemně komunikujících avšak jinak samostatných gossipových aplikací.

Dále by framework měl podporovat simulaci nejrůznějších chyb a poruch. Typickými chybami, se kterými se ve fyzických realizacích gossipových systémů patrně budeme setkávat, jsou ztráty zpráv a poruchy uzlů. Gossipové systémy (jakožto systémy využívající kombinace emergence a sebeorganizace) by měly být vůči takovýmto chybám odolné. Framework by měl umožnit takovéto chyby simulovat a tím uživateli umožnit posuzovat míru odolnosti systému vůči různým množstvím výskytů takovýchto chyb.

Vedle popisu experimentů prostřednictvím skriptů by měl framework podporovat i interaktivní simulaci – měl by uživateli dovolit spustit simulaci a v reálném čase do ní zasahovat, tj. zkoumat stav simulovaného systému, simulovat vstupy z vnějšího prostředí, atp.

Vzhledem k tomu, že fyzické realizace gossipových systémů se skládají z velkého množství paralelně běžících uzlů, bude simulace takovéhoto systému relativně snadno paralelizovatelná. Je proto vhodné, aby simulátor umožňoval rozdělení simulační aplikace na množství konkurentních (a tudíž na vhodném stroji potenciálně paralelních) vláken.

Splnit poslední dva požadavky – interaktivní simulaci a důraz na paralelizovatelnost – je v běžně používaných programovacích jazycích jako Java nebo C++ dosti náročné. Nicméně existují jazyky, pomocí kterých je možné tyto požadavky zajistit relativně snadno. Pravděpodobně nejznámější je programovací jazyk Erlang.

5.1.1 Erlang

Erlang je interpretovaný dynamicky typovaný programovací jazyk zaměřený zejména na programování konkurentních a distribuovaných systémů. Jazyk byl vyvinut firmami Ericsson a Ellemtel (Ericsson and Ellemtel Computer Science Laboratories) jako prostředek pro vytváření převážně síťových a telekomunikačních aplikací, nicméně našel uplatnění i v mnoha dalších oblastech, mj. v mnoha *soft real-time* systémech [3].

Erlang byl pojmenován podle dánského matematika A. K. Erlanga, známého zejména díky svým pracem, které se zabývaly statistickou analýzou komunikačních systémů. Alternativně lze také název Erlang vnímat jako zkratku z *Ericsson language* [19].

Podstatnou vlastností Erlangu je, že byl navržen jako jazyk umožňující explicitně vyjadřovat konkurentnost. Program v Erlangu se skládá z množiny dynamicky vytvářených a rušených procesů vzájemně komunikujcích prostřednictvím zasílání zpráv. Procesy jsou zde jazykový konstrukt, typicky nejsou nijak přímo závislé na operačním systému (nebývají totožné s procesy či vlákny operačního systému), veškerou správu procesů zajišťuje virtuální stroj [1]. Prakticky všechny v současné době používané implementace virtuálního stroje

Erlangu obsahují podporuju pro symetrický multiprocessing – ta umožňuje velice efektivní paralelizaci běhu programu na v současnosti běžných vícejaderných procesorech [2].

Při návrhu jazyka se autoři soustředili na co největší efektivitu jak správy procesů (jejich vytváření, udržování a rušení) tak i komunikace mezi procesy. V důsledku toho jsou procesy Erlangu jsou velmi lehké (*light weight*) – vytváření a spravování procesů je velice nenáročné jak z hlediska paměťových nároků tak i z pohledu časové náročnosti [1]. Erlang umožňuje vytváření i velmi velkého množství procesů. Jednotlivé procesy je možné propojit (*link*) a tím umožnit vzájemné sledování jejich běhu.

V Erlangu je použit model konkurentnosti, který využívá zasílání zpráv. Neexistuje zde paměť sdílená mezi procesy – paměťové prostory jednotlivých procesů jsou zcela odděleny. Veškerá komunikace mezi procesy je možná pouze prostřednictvím asynchronního zasílání zpráv. Podle autorů jazyka je hlavní důvod pro tuto architekturu ten, že jakékoli sdílení paměti vede k řadě problémů – například pokud procesy sdílejí data, je celkem komplikované zajistit, aby každý z nich mohl běžet na samostatném počítači; další problém je, že přístup k datům prostřednictvím kritických sekcí vede často ke zbytečné synchronizaci mezi procesy [1]. Všechny tyto vlastnosti konkurentnosti se sdílenou pamětí vedou ke snížení efektivity takovýchto systémů.

Další vlastností Erlangu je, že explicitně podporuje distribuovaný běh aplikací mezi několika počítači. Díky tomu, že mezi procesy neexistuje sdílená paměť, je zajištění distribuovanosti programu relativně snadné – v podstatě stačí alokovat jednotlivé paralelní procesy na různé počítače [1].

Ačkoli je jádro Erlangu v zásadě funkcionální, není možno tento jazyk klasifikovat jako plně funkcionální jazyk – podporuje totiž operace s vedlejšími efekty, které čistě funkcionální paradigma zavrhuje [19]. Nejvýznamnější operací s vedlejším efektem v Erlangu je zaslání zprávy – tedy samotný základní koncept komunikace mezi procesy. Vedle toho existují v Erlangu i některé další (méně významné a typicky samotnými autory nedoporučované) operace s vedlejšími efekty, jako například operace se slovníkem procesu [3]. U typických aplikací platí, že jednotlivé procesy jsou (pokus se programátor vyhne jistým nedoporučovaných operacím) programovány čistě funkcionálně, zatímco program jako celek za funkcionální považovat nelze [19].

Erlang je jazyk dynamicky typovaný. V jazyce je dostupné množství jednoduchých datových typů včetně celých čísel, reálných čísel (čísla s plovoucí řádovou čárkou), atomů, identifikátorů procesů, a dalších. Atom je zde chápán obdobně jako v jazyce Prolog – slouží pro reprezentaci nečíselných konstantních hodnot [2]. Podporované složené datové typy jsou n-tice (*tuple*) a seznamy.

Kód programu napsaného v Erlangu se skládá z jednoho či více modulů. Každý modul má unikátní jméno (datového typu atom) [19].

Kód jednotlivých modulů je typicky kompilován do *byte code*, který je následně interpretován emulátorem virtuálního stroje [19]. Součástí virtuálního stroje je i *garbage collector* zajišťující správu paměti – použití garbage collectoru u jazyka určeného pro real-time systémy se může jevit jako kontroverzní, nicméně v praxi se ukazuje, že aplikace vytvořené v Erlangu jsou i přes toto rozhodnutí velmi výkonné [8].

Jednou z výhod Erlangu, která je často využívána např. v síťových aplikacích, je, že umožňuje rekompilaci programu za jeho běhu – je možné upravit kód programu a dané změny uvést do provozu, aniž by bylo nutné danou aplikaci restartovat (*code hot-swapping*). S běžící aplikací je možné interagovat prostřednictvím Erlang shellu [19].

Obecně je Erlang vhodný pro psaní programů skládajících se z velkého množství konkurentních vláken, které na sobě vzájemně nejsou příliš závislé a komunikují spolu prostřed-

nictvím relativně krátkých zpráv – takovéto programy dokáže virtuální stroj interpretovat velice efektivně, a to zejména na počítačích podporujících symetrický multiprocessing [2]. Právě tato vlastnost – vysoká efektivita práce i s velkým množstvím vzájemně nepříliš závislých procesů – dělá z tohoto jazyka vhodný prostředek pro simulaci aplikací Gossipu.

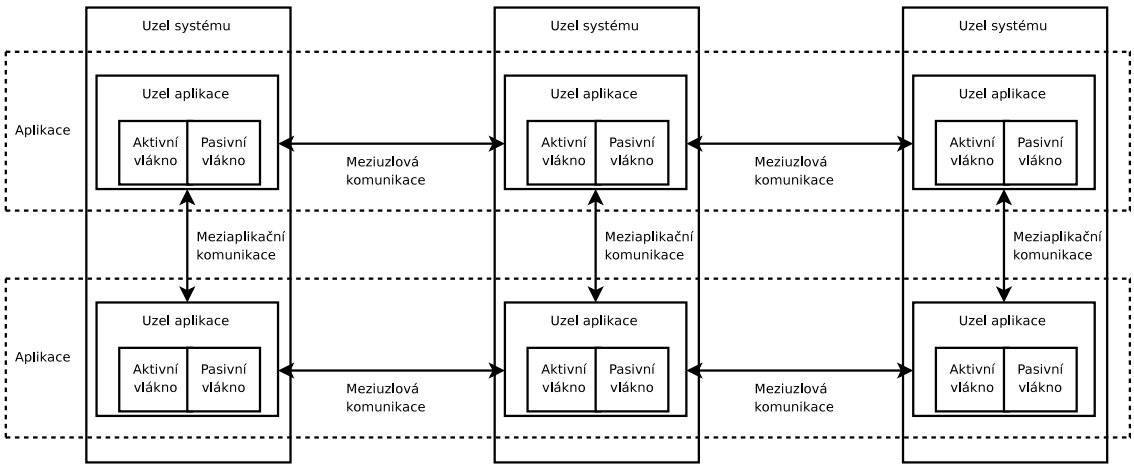
5.1.2 Reprezentace gossipového systému

Jak už bylo zmíněno dříve, gossipový systém je popsán množinou vzájemně komunikujících gossipových aplikací. Systém je tvořen množinou uzlů. Na každém z těchto uzlů systému běží pro každou aplikaci jedna instance uzlu aplikace. Každý uzel jedné aplikace se skládá ze dvou vláken – aktivního a pasivního. Mezi těmito vlákny je sdílen stav tohoto uzlu aplikace – obě vlákna jej mohou číst a (alespoň v případě varianty *push-pull*) také měnit.

Mezi uzly aplikace dochází ke dvěma druhům komunikace:

- v rámci uzlu systému, kdy spolu komunikují uzly náležející jednomu uzlu systému, ale různým aplikacím (tj. meziaplikační komunikace)
- v rámci aplikace, kdy spolu komunikují uzly patřící ke stejné aplikaci, ale k různým uzlům systému (tj. meziuzlová komunikace)

Možností, jak takovýto ne zcela jednoduchý systém modelovat procesy jazyka Erlang, je relativně velké množství. Obecně se řešení, kterými jsem se zabýval, liší ve dvou bodech. První z těchto odlišností je, zda je činnost aktivního a pasivního vlákna vykonávána jediným procesem, nebo je každé z těchto vláken reprezentováno procesem zvláštním. Dále se tato řešení liší tím, zda a jak jsou činnosti jednotlivých aplikací rozděleny mezi více procesy.



Obrázek 5.1: Schéma gossipového systému. Je zde vyznačen vztah mezi uzly systému, uzly aplikace a aktivním a pasivním vláknem, dále meziuzlová a meziaplikační komunikace.

Oddělená vlákna a oddělené aplikace

První z možností, která se nabízí, je přiřadit každému z obou vláken každého uzlu každé aplikace jeden proces. V tomto případě by byl systém sestávající z n uzlů a m aplikací reprezentován $2 \cdot m \cdot n$ procesy. Meziaplikační i meziuzlovou komunikaci je v takovémto případě

možné nepříliš komplikovaně realizovat prostřednictvím zasílání zpráv mezi odpovídajícími procesy.

Problém je však stav uzlu aplikace – ten je totiž sdílen dvěma samostatnými procesy, které reprezentují aktivní a pasivní vlákno téhož uzlu téže aplikace. Vzhledem k tomu, že Erlang nepodporuje sdílení paměti mezi procesy, je nutné implementovat toto sdílení stavu prostřednictvím asynchronního zasílání zpráv.

Oba procesy (tj. procesy pro aktivní a pasivní vlákno téhož uzlu aplikace) musejí mít vždy přístup k *aktuální* hodnotě stavu a v případě (patrně nejčastěji používané) varianty Gossipu *push-pull* mohou také oba procesy tento stav měnit. Zajistit takovéto chování prostřednictvím asynchronního zasílání zpráv je zřejmě dosti komplikované a výsledné řešení by bylo značně neefektivní.

Je potřeba si uvědomit, že u varianty *push-pull* dochází v každém cyklu u každého z obou vláken průměrně k jednomu volání update(). Řekněme, že stav udržovaný na obou procesech reprezentujících jednotlivá vlákna je aktuální. Pokud jeden proces způsobí změnu stavu, pošle tuto informaci automaticky druhému. V tomto případě by se snadno mohlo stát, že oba uzly provedou aktualizaci stavu naráz – což by vedlo ke dvěma platným „aktuálním“ stavům jednoho uzlu, z nichž jeden by následně musel být odstraněn ve prospěch druhého. Takovéto chování by zcela jistě nebylo žádoucí – docházelo by tak velmi často ke ztrátám informace v systému.

Pokud by však byly informace o aktuálním stavu udržovány jen na jednom z obou procesů, musel by ten druhý s tímto procesem konzultovat každý přístup k aktuální hodnotě stavu – docházelo by velice často k synchronní komunikaci mezi procesy. Takovýto přístup by zřejmě vedl k podstatnému snížení výkonnosti celého systému.

Sdílená vlákna a oddělené aplikace

Dále se nabízí možnost reprezentovat aktivní i pasivní vlákno jednoho uzlu jedné aplikace dohromady jediným procesem Erlangu. Systém skládající se z n uzlů a m aplikací by v tomto případě byl reprezentován $m \cdot n$ procesy Erlangu. Problém tohoto řešení je, jak vhodně zajistit, aby jediný proces Erlangu prováděl chování obou vzájemně nezávislých vláken naráz.

Chování obou vláken je popsáno v části 3.1. Pasivní vlákno prakticky jen čeká na obdržení zprávy a na jednotlivé příchozí zprávy reaguje. V rámci reakce na jednu příchozí zprávu nesmí dojít k zablokování běhu vlákna v důsledku čekání na zprávu jinou. Aktivní vlákno pouze periodicky spouští relativně jednoduchý kód. Mezi běhy tohoto kódu je vlákno neaktivní. Pokud by kód opakovaně spouštěný aktivním vláknem nemohl způsobit zablokování procesu, bylo by možné chování obou vláken propojit v jednom procesu tak, že by během čekání (tj. neaktivity) chování odpovídajícího aktivnímu vláknu docházelo ze zpracování příchozích zpráv podle chování pasivního vlákna. Nepříjemné je, že v případě *pull* a *push-pull* varianty Gossipu dochází k zablokování aktivního vlákna během čekání na odpověď od cizího uzlu. Je tedy nutné i během tohoto čekání reagovat na příchozí zprávy od ostatních uzlů.

Z pohledu chování aktivního vlákna by tedy u takovéhoto řešení bylo možno reagovat na příchozí zprávu pro pasivní vlákno ve dvou momentech: v průběhu vykonávání funkce wait() nebo (v případě variant *pull* nebo *push-pull*) v průběhu čekání na odpověď od komunikačního partnera.

Je zřejmé, že takovéto řešení neodpovídá zcela přesně chování uzlu aplikace – pokud přijde od cizího uzlu zpráva v průběhu provádění chování aktivního vlákna (tj. v průběhu

výpočtu funkce selectPeer() nebo update() aktivního vlákna), nebude moci pasivní chování okamžitě zareagovat na tuto příchozí zprávu. Avšak ani v realizaci gossipového systému odpovídajícího přesně kostře popsané v části 3.1 (tedy využívajícího sdílené paměti mezi oběma vlákny) není vždy možná naprosto okamžitá odezva pasivního vlákna – pasivní vlákno může být zaneprázdněno odpovídáním na zprávu od jiného uzlu. Navíc, pokud nebudou funkce selectPeer() ani update() příliš výpočetně náročné, bude zvýšení doby odezvy uzlu tohoto řešení ve srovnání s ideálním řešením zanedbatelné.

Dalším problémem tohoto řešení je meziaplikační komunikace – je nutné zajistit možnost komunikace mezi všemi procesy odpovídajícími jednomu uzlu systému (tj. mezi odpovídajícími si uzly všech aplikací) prostřednictvím pravidelného zasílání zpráv s aktuální hodnotou stavu daného uzlu dané aplikace.

Otázkou je, jak často tyto zprávy realizující meziaplikační komunikaci zasílat. Jednou z množností by bylo zasílat meziaplikační zprávu po každé změně stavu odesílajícího uzlu aplikace. Nicméně ve většině případů přijímající uzel aplikace nepotřebuje znát naprosto současný stav ostatních aplikací, na kterých závisí – typické použití meziaplikační komunikace je takové, kdy funkční aplikace využívá stav aplikace pro udržování topologie – např. aplikace Peer sampling, která stejně reaguje na faktické změny systému (tj. výpadky uzlů) někdy i s nezanedbatelným zpožděním.

Tyto meziaplikační zprávy tedy stačí rozesílat v pravidelných intervalech, například jednou v každém cyklu, bez ohledu na reálné změny stavu. S četností rozesílání takovýchto zpráv mezi různými aplikacemi je potom možné manipulovat pomocí nastavení různých délek cyklů (tj. různých funkcí wait()) pro jednotlivé aplikace. Může se například stát, že některá funkční aplikace závislá na aplikaci pro udržování proměnlivé topologie bude vyžadovat v každém svém cyklu znalost jiné množiny sousedů. Toto je možné zajistit s vysokou pravděpodobností pomocí nastavení výrazně větší průměrné délky cyklu dané funkční aplikace v porovnání s délkou cyklu aplikace pro udržování topologie.

Výměnu meziaplikačních zpráv je možné realizovat tak, že uzly každé aplikace budou posílat své stavy odpovídajícím uzlům všech ostatních aplikací. Problémem v tomto případě je značné množství takto rozeslaných zpráv při velkém množství aplikací v systému. Pokud máme v systému m aplikací, pak (předpokládáme-li stejnou délku cyklu u všech aplikací) bude v rámci každého uzlu systému v každém cyklu rozesláno $2 \cdot \binom{m}{2}$ zpráv (to lze ekvivalentně zapsat jako $m \cdot (m-1)$, počet zpráv rozeslaných v jednom cyklu by tedy rostl kvadraticky vůči počtu aplikací). Je tedy vhodnější rozesílat meziaplikační zprávy jen v tom směru, ve kterém je to potřeba, tj. mezi těmi aplikacemi, které jsou na sobě závislé. V tomto případě bude nutné, aby uživatel při definici systému uvedl pro každou aplikaci informaci, na kterých ostatních aplikacích tato aplikace závisí (tedy, od kterých aplikací má tato aplikace přijímat stav).

Sdílená vlákna a sdílené aplikace

Třetí možnost, jak reprezentovat gossipový systém prostřednictvím procesů Erlangu, je reprezentovat jedním procesem celý uzel systému. V tomto případě by jeden proces realizoval běh aktivního i pasivního vlákna všech aplikací běžících na jednom uzlu systému. Systém sestávající z n uzlů by tedy byl reprezentován n procesy bez ohledu na počet aplikací.

Při implementaci tohoto řešení by kód reprezentující jeden uzel jedné aplikace mohl být v podstatě stejný jako v případě předchozím. Avšak jeden proces by musel střídavě vykonávat činnosti všech těchto aplikací. Jednoduché řešení může vycházet z předpokladu, že všechny aplikace běžící na jednom uzlu systému mají stejnou délku cyklu – běží tedy

synchronně. V tomto případě by všechny aplikace sdílely jedinou funkci wait(). Zde je problém u variant Gossipu *pull* a *push-pull*, kde je délka cyklu ovlivněna nejen funkcí wait(), ale i průměrnou délkou čekání na odpověď od komunikačního partnera. Pokud bude tato doba u různých aplikací rozdílná (např. v důsledku rozdílné výpočetní náročnosti funkcí update(); u varianty *push* je tato doba navíc vždy nulová), budou rychleji běžící aplikace negativně ovlivňovány těmi pomalejšími.

Alternativní řešení, kdy jednotlivé aplikace v rámci jednoho procesu mají různé funkce wait() a běží zcela asynchronně, by připomínalo realizaci multitaskingu na běžných jednoprocesorových počítačích – bylo by dosti komplikované, navíc v jazyce Erlang, který umožňuje snadné a velice efektivní vytváření nových procesů, by byla realizace konkurentnosti prostřednictvím takovéto obdoby multitaskingu zcela proti zásadám jazyka a působila by spíše absurdně.

Výhodou synchronní varianty tohoto řešení by byla velice jednoduchá meziaplikační komunikace – na každém uzlu systému by byly vždy přítomny aktuální stavy všech aplikací, není tedy nutné stavy v ramci jednoho uzlu neustále přeposílat jako v případě předchozím. Nicméně toto řešení má i řadu nevýhod. Chování procesu je zde relativně komplikované. Hlavní nevýhodou je však už zmíněná skutečnost, že aplikace běžící různou rychlostí vzájemně ovlivňují svůj běh. V důsledku toho se může chování některých aplikací i dosti vzdalovat chování ideální implementace gossipové aplikace.

Oddělená vlákna a sdílené aplikace

Poslední variantou je případ, kdy jsou odděleny procesy pro pasivní a aktivní vlákno, avšak v rámci jednoho procesu jsou realizovány všechny aplikace. Tedy jeden proces reprezentuje pro jeden uzel buď pouze aktivní nebo pouze pasivní vlákna všech aplikací. V takovémto případě by byl systém skládající se z n uzlů reprezentován $2 \cdot n$ procesy bez ohledu na počet aplikací.

Podobně jako u první varianty je zde problém se zajištěním toho, aby měly oba procesy realizující vlákna jednoho uzlu přístup v tomto případě ke všem aktuálním stavům všech aplikací daného uzlu.

Dalším problémem je, podobně jako ve třetí variantě, zajištění běhu různých aplikací v rámci jednoho procesu. U procesu realizujícího pasivní vlákna všech aplikací by stačilo zajistit, aby byl schopen reagovat na všechny meziuzlové zprávy pro všechny aplikace. Situace u procesu pro aktivní vlákna všech aplikací je situace komplikovanější a v podstatě obdobná situaci popsané u varianty se sdílenými vlákny a sdílenými aplikacemi.

Tato varianta tedy v podstatě pouze kombinuje nevýhody první a třetí varianty, aniž by přinášela jakoukoli výhodu.

Realizace prostřednictvím procesů

Z těchto čtyř způsobů, jak modelovat gossipový systém procesy Erlangu, jsem se rozhodl realizovat variantu se sdílenými vlákny a oddělenými aplikacemi – zdá se, že tato varianta představuje nejvhodnější kompromis mezi efektivností běhu a přesností simulace. Každý proces Erlangu tedy bude reprezentovat jeden uzel jedné aplikace, v rámci tohoto procesu poběží obě vlákna daného uzlu.

Takováto simulace gossipového systému bude realizována prostřednictvím velkého množství procesů. Ačkoli je chování jednoho procesu typicky relativně jednoduché, dá se předpokládat, že celková výpočetní náročnost simulace bude dosti značná. Mohlo by se tudíž zdát, že by bylo výhodné rozdělit simulaci mezi několik počítačů. Toto rozdělení by bylo

možno realizovat relativně snadno – Erlang velmi dobře podporuje psaní takovýchto distribuovaných programů.

Nemyslím však, že by takovéto distribuování simulace bylo vhodné. Předpokládejme pro jednoduchost simulaci gossipového systému skládajícího se z n uzlů s jedinou aplikací komunikující způsobem *push*. Nechť tato aplikace vyžaduje pro každý uzel výběr náhodného komunikačního partnera s rovnoměrným rozdělením pravděpodobnosti (což je asi nejtypičtější případ způsobu výběru komunikačního partnera). Pokud bychom takovouto simulaci rovnoměrně distribuovali mezi k počítačů, docházelo by *v každém cyklu systému* k zaslání průměrně $\frac{k-1}{k}n$ zpráv Gossipu po počítačové síti. Pouze $\frac{1}{k}n$ zpráv by bylo v daném cyklu zasláno procesům běžícím na stejném počítači. Množství informace přenášené po síti by tudíž bylo značné. Vzhledem k tomu, že v případě přenosu po síti se dá předpokládat zpoždění zpráv značně větší než v případě komunikace mezi procesy v rámci jednoho počítače, mělo by distribuování simulace negativní vliv na běh celého simulovaného systému.

5.1.3 Identifikace procesů

Zatím jsem zjednodušeně předpokládal, že gossipový systém je plně popsán počtem svých uzlů a množinou plně popsaných aplikací. Platí, že každý uzel systému se skládá z množiny uzlů aplikací – pro každou aplikaci je v rámci jednoho uzlu systému jeden aplikační uzel (vizte úvod kapitoly 5.1.2). Při způsobu reprezentace uzlu, který jsem zvolil, dále platí, že každý uzel každé aplikace je reprezentován jedním procesem Erlangu. Jeden proces reprezentuje obě vlákna tohoto uzlu aplikace.

Nicméně počet uzlů systému obecně nemusí být konstantní – v průběhu simulace se může měnit. Uzly mohou do systému přibývat a ze systému odcházet z různých důvodů, mj. z důvodů poruchy uzlu resp. jejího odstranění, další důvody přibývání a mizení uzlů mohou být aplikačně závislé – např. připojení a odpojení uživatele ze systému v případě simulace některých síťových aplikací.

Dalším problémem je, že uživatel frameworku bude pravděpodobně v průběhu simulace chtít zkoumat stav libovolného uzlu. Tudíž mu musí být umožněno přistoupit k libovolnému procesu reprezentujícímu nějaký uzel aplikace.

Každý proces Erlangu je reprezentován svým identifikátorem (PID). Vedle těchto identifikátorů umožňuje Erlang své procesy registrovat – přiřadit jim atom, který pak může sloužit jako zástupce identifikátoru.

Protože množství procesů vytvořených při simulaci gossipového systému bude dosti značné, není příliš výhodné udržovat nějak centralizovaně seznam (či jinou datovou strukturu) jejich identifikátorů – takováto databáze by totiž tvořila úzké místo simulačního systému.

Jistě je možné nijak neudržovat žádné informace o existujících procesech mimo ně samotné. Procesy reprezentující jeden uzel by samozřejmě měly mít informace o sobě navzájem, alespoň pro potřeby meziaplikační komunikace, nicméně informace o adresách procesů pro meziuzlovou komunikaci lze zajistit pomocí aplikace Peer sampling (vizte část 4.2), tj. v samotné aplikaci Gossipu. Nevýhoda takovéhoto řešení spočívá v relativně velké výpočetní náročnosti aplikace Peer sampling (její funkce update() provádí komplikovanější množinové operace). V tomto případě by totiž *každá* simulace musela nutně obsahovat tuto aplikaci. A to včetně jednoduchých simulací, u kterých není požadována velká přesnost – v takovýchto simulacích by pak realizace aplikace Peer sampling (tj. spíše pomocné aplikace s pouze nepřímým vlivem na výsledek simulace) tvořila výpočetně nejsložitější část simulace. Další nevýhodou tohoto řešení je, že neumožňuje uživateli přistoupit kdykoli ke kterémukoli uzlu systému – uživatel by si v tomto případě nejspíše udržoval jen znalost

identifikátorů relativně malé podmnožiny procesů.

Domnívám se tedy, že je výhodnější řešení spočívající v registraci každého procesu reprezentujícího uzel nějaké aplikace pod unikátním jménem. Toto jméno bude atom, jehož forma bude jednoznačně odvozena od kombinace čísla uzlu systému a od označení aplikace – tyto dva údaje tvoří jednoznačnou identifikaci daného procesu.

V důsledku toho musí mít každá aplikace přiřazeno své unikátní označení – tedy jméno (toto jméno musí být unikátní pouze v rámci simulovaného systému). A dále každý uzel systému musí mít přiřazeno unikátní číslo z rozmezí $1, 2, \ldots n$, kde n je celkový počet těchto uzlů.

Potenciálním problémem tohoto přístupu je fakt, že v současných implementacích Erlangu je omezeno maximální množství procesů i atomů existujících v systému [12]. Protože je každý uzel každé aplikace simulovaného systému reprezentován jedním procesem a všechny tyto procesy jsou pojmenovány prostřednictvím atomů, mohou tato omezení počtu procesů a atomů omezit maximální počet uzlů, který je framework schopen simulovat. Nicméně v současných implementacích Erlangu je možné výchozí maximální počet procesů i atomů navýšit [12] a horní limit tohoto počtu je natolik velký, že by tato omezení neměla mít na možnosti frameworku praktický vliv.

Poruchy uzlů, opravy poruch, odcházení uzlů ze systému a objevování se nových uzlů je možné simulovat pomocí „vypínání a zapínání uzlů“ – každý uzel bude mít možnost být převeden do speciálního neaktivního stavu, který reprezentuje jeho poruchu, výpadek či zánik, a případně zase zpět do stavu normálního (simulace opravy poruchy či vzniku nového uzlu). V tomto případě značí číslo n maximální počet funkčních uzlů v systému. Uživatel tedy nebude specifikovat přesný počet uzlů v systému, ale jejich maximální počet. Přesný počet uzlů v systému v daný okamžik bude odvozen od maximálního počtu uzlů v systému a počtu momentálně „vypnutých“ uzlů.

Výhodou tohoto přístupu je to, že se simulace skládá vždy z pevného počtu procesů. To značně zjednodušuje návrh – není nutné řešit výjimky při pokusu o komunikaci s neexistujícím procesem, navíc vznik a zánik uzlů bude realizován extrémně jednoduše. Procesy reprezentující „vypnutý“ uzel nebudou vyvíjet prakticky žádnou činnost (budou pouze čekat na přijetí speciální zprávy od uživatele o tom, že se mají zapnout), nebudou tedy představovat nějakou problematickou zátěž systému.

Další výhodou je, že uživatel bude moci kdykoli v průběhu simulace snadno přistoupit k jakémukoli uzlu jakékoli aplikace – pouze si pro tento účel musí odvodit jméno procesu reprezentujícího daný uzel na základě čísla uzlu systému a jména aplikace.

Dále je díky tomuto přístupu možné velice jednoduše realizovat funkci selectPeer() pro jednoduché simulace – pro případy, kdy není vhodné zbytečně komplikovat simulaci přidáváním aplikace Peer sampling. V takovémto případě je možné realizovat funkci selectPeer() pouze prostřednictvím výběru čísla uzlu jako pseudonáhodného čísla vybraného s rovnoměrným rozložením pravděpodobnosti z rozmezí $1, \ldots, n$.

5.1.4 Zadávání gossipového systému

Při modelování gossipového systému by měl uživatel specifikovat chování všech aplikací a maximální počet uzlů v systému. Definice chování aplikace se skládá z následujících prvků:

- jméno aplikace
- definice funkce update()
- definice funkce selectPeer()

- definice varianty protokolu Gossipu (tedy *push*, *pull* nebo *push-pull*)
- definice inicializační funkce
- definice funkce wait()
- určení, na kterých aplikacích je tato aplikace závislá

Jméno aplikace je nutné jednak pro pojmenování jednotlivých procesů (kap. 5.1.3), jednak pro zadání závislostí mezi aplikacemi za účelem určení směru meziaplikační komunikace.

Funkce update() by měla mít v tomto případě tři parametry – kromě aktuálního stavu daného uzlu a stavu jeho komunikačního partnera by měla mít k dispozici i výsledky meziaplikační komunikace – seznam stavů všech ostatních aplikací daného uzlu systému.

Podobně funkce selectPeer() musí být ovlivněna stavy všech aplikací daného uzlu systému – stavem své aplikace i stavy ostatních aplikací. Například v případě aplikace Peer sampling pracuje funkce selectPeer() s aktuálním stavem své vlastní aplikace, zatímco u funkčních aplikací závislých na Peer sampling je funkce selectPeer() závislá na aktuálním stavu právě aplikace Peer sampling – tedy na stavu jiné aplikace téhož uzlu systému.

Součástí definice aplikace musí být i funkce, která pro daný uzel aplikace vrátí jeho počáteční stav – tedy inicializační funkce. Jedinou informací, kterou tato funkce může o daném uzlu znát, je jeho číslo. Inicializace je závislá na konkrétní aplikaci i na potřebách a účelu simulace.

Protože jsem se rozhodl realizovat variantu reprezentace gossipového systému se sdílenými vlákny a oddělenými aplikacemi, je součástí definice každé aplikace i její funkce wait(). Zde je tato funkce chápána tak, že vrací číselnou hodnotu, která udává dobu, po kterou má být aktivní část procesu v rámci jednoho cyklu v nečinnosti. Nejedná se tedy o funkci, která by měla způsobit zablokování běhu aktivního vlákna po určitou dobu, funkce pouze určuje délku tohoto zablokování.

Podstatnou součástí definice aplikace je i seznam jmen těch cizích aplikací, na jejichž stavu je daná aplikace závislá. Tento seznam by měl sloužit pro určení, ve kterém směru by mělo docházet k meziaplikační komunikaci. Každá aplikace potřebuje mít informace o tom, od kterých aplikací má přijímat meziaplikační zprávy a kterým aplikacím má meziaplikační zprávy odesílat. Uživatel přímo zadá pouze informaci, od kterých aplikací daná aplikace meziaplikační zprávy přijímá. Informace o automatickém odesílání meziapliačních zpráv bude od této informace automaticky odvozena při vytváření simulace.

Na rozdíl od výčtu uvedeného v úvodu do části 5.1 zde není uvedena nutnost specifikovat datový typ stavu uzlu. Erlang je totiž dynamicky typovaný jazyk a tuto explicitní specifikaci datového typu nevyžaduje. Je pouze nutné, aby funkce update() vracela jako svůj výsledek nový stav, který je stejného datového typu, jako jsou stavy, které tato funkce bere za své parametry.

Alternativní způsob zadávání systému

Vedle (preferovaného) způsobu zadávání aplikace Gossipu, popsaného v předchozí části, by framework měl umožňovat i zadávání aplikace na podstatně primitivnější úrovni. V tom případě by se definice aplikace měla skládat z těchto prvků:

- jméno aplikace
- definice funkce active()

- definice funkce passive()
- definice inicializační funkce
- určení aplikací, na kterých je tato aplikace závislá

Jméno aplikace, inicializační funkce a seznam závislostí této aplikace jsou stejné jako v předchozím případě.

Funkce active() slouží pro specifikaci kódu jednoho cyklu kódu aktivního vlákna daného uzlu, avšak bez čekání v důsledku volání funkce wait(). Jedná se o funkci, která bude daným procesem volána v pravidelných intervalech – jednou v každém cyklu. Funkce by neměla zablokovat běh uzlu v důsledku čekání na odpověď od jiného stavu. Návratová hodnota funkce je ignorována.

Funkce passive() slouží pro specifikaci funkce pasivního vlákna daného uzlu prováděné po dobu jednoho cyklu. Funkce by měla čekat na příchozí komunikaci od cizích uzlů. Doba čekání však musí být omezená. V ideálním případě by se měla funkce provádět právě tak dlouho, jaká je požadovaná délka jednoho cyklu – doba provádění této funkce přímo určuje délku cyklu. Funkce vrací hodnotu nového stavu systému. Pokud stav nebyl v průběhu funkce změněn, měla by funkce vracet jeho starou hodnotu.

Obecný pseudokód práce procesu reprezentujícího jeden uzel jedné aplikace:

```
1: while 1
3:       active(state, otherAppStates)
2:       state := passive(state, otherAppStates)
4:       userMessages(state)
5:       otherAppStates := interAppMessages(state, otherAppStates)
6: end
```

Proměnná *state* značí stav daného uzlu dané aplikace, *otherAppStates* značí seznam stavů těch aplikací, od kterých daná aplikace přijímá meziaplikační zprávy. Funkce userMessages() a interAppMessages() slouží pro zajištění komunikace s uživatelem resp. pro meziaplikační komunikaci. Tyto funkce jsou zajištěny simulačním systémem.

Podstatné z daného pseudokódu je, že funkce passive() musí provádět svůj kód, tj. čekání na příchozí zprávy a reakce na jejich případná přijetí, pouze omezenou dobu. Protože je funkce passive() jedinou funkcí, která provádí čekání na příchozí zprávy, průměrná doba tohoto čekání přímo určuje velikost cyklu. Pokud bude funkce čekat neomezenou dobu, způsobí, že ostatní funkce nebudou nikdy spouštěny.

Podobně by funkce active() vůbec neměla provádět čekání na příchozí zprávy. Proto ani nemá možnost měnit stav uzlu.

Z pseudokódu je zřejmé, že obě funkce (tj. active() i passive()) jsou spouštěny stejně často a za sebou. Mohlo by se zdát, že by bylo výhodné mít místo těchto dvou funkcí jen jednu, která by prováděla funkčnost obou. V zásadě uživateli nic nebrání to takto vyřešit – definovat funkci active() jako prázdnou funkci (tj. vracející konstantu, která je stejně ignorována) a v rámci passive() definovat celé chování uzlu.

Rozdíl mezi funkcemi active() a passive() je v jejich sémantice. Funkce active() reprezentuje funkci aktivního vlákna uzlu – měla by vybírat komunikační partnery a navazovat s nimi komunikaci, funkce passive() zase reprezentuje pasivní vlákno uzlu – jeho úkolem je přijímat zprávy od ostatních uzlů a reagovat na ně.

Funkce passive() by neměla generovat žádné jiné zprávy, než ty zaslané v rámci okamžité reakce na zprávy příchozí. V případě zasílání zpráv jako reakce na zprávy jiné je třeba dávat pozor na to, aby v rámci zpráv zasílaných funkcemi passive() různých uzlů nemohlo dojít k zacyklení. Důvod pro tento požadavek je ten, že běh aplikace, jejíž funkce passive() generuje zprávy jinak než v rámci reakce na zprávy zaslané funkcí active(), by nebylo možno zastavit – tedy dočasně zastavit vývoj aplikace pro účely statického zkoumání jejího stavu. Samozřejmě, je-li uživatel ochoten zkoumat stav aplikace pouze za jejího běhu, je možné z tohoto požadavku na negenerování zpráv funkcí passive() slevit.

Je zřejmé, že specifikace aplikace prostřednictvím funkcí active() a passive() je ta podstatně náročnější a méně přirozená varianta. Uživatel musí zajistit prakticky vše – zasílání meziuzlových zpráv, jejich přijímání, musí dát pozor, aby nedocházelo k hromadění zpráv ve schránkách, musí zajistit, aby funkce passive() nemohla běžet neomezenou dobu, musí vzít v úvahu možnost deadlocku při nevhodně navržené komunikaci mezi uzly, atp.

Varianta zadávání aplikace prostřednictvím funkcí update(), selectPeer(), wait(), atd., by tedy měla být preferována. Nicméně existují případy, kdy tuto přirozenější a jednodušší variantu využít nelze – to je hlavní důvod existence možnosti definice chování uzlů prostřednictvím zadání funkcí active() a passive().

5.1.5 Komunikace s uživatelem

Vzhledem k tomu, že by systém měl poskytovat možnost interaktivní simulace, musejí být procesy reprezentující jednotlivé uzly aplikací schopny reagovat na povely od uživatele. Zejména mu musejí být schopny sdělit svůj aktuální stav – hlavním cílem simulace zřejmě bude zkoumání vývoje stavu jednotlivých uzlů.

Uživateli musí být dále zpřístupněna možnost za běhu vypínat a zapínat procesy a zjišťovat, které uzly jsou právě vypnuté či zapnuté.

Dále musí být uživatel schopen zastavovat a znovu spouštět běh celé simulace. Musí také existovat možnost spustit systém jen na omezenou dobu – například aby všechny uzly provedly daný počet cyklů a zastavily se.

Rozdíl mezi zastavováním (resp. spouštěním) procesů a jejich vypínáním (resp. zapínáním) je ten, že účelem vypnutí je simulovat poruchu či konec existence daného uzlu, zatímco zastavením uzlu pouze zamezíme jeho dalšímu vývoji (typicky dává smysl zastavovat pouze všechny uzly naráz). U zastaveného uzlu jsme schopni zkoumat jeho stav. V případě porouchaného či neexistujícího (tedy vypnutého) uzlu pojem stavu nedává příliš velký smysl – tedy stav je nedostupný. Zastavený uzel sice negeneruje žádnou činnost, avšak je schopen pasivně reagovat na komunikaci jak od ostatních uzlů a aplikací, tak i z vnějšku. Vypnutý uzel ignoruje veškerou meziaplikační i meziuzlovou komunikaci.

Spuštění a zastavení celé simulace naráz je možno realizovat prostým odesláním určité speciální zprávy všem procesům reprezentujícím jednotlivé uzly aplikací. Obdobně je možné realizovat i spouštění simulace pro určitý počet cyklů.

Operaci spuštění simulace pro pevný počet cyklů je možné využít i pro psaní skriptů popisujících simulační experimenty. Typický popis experimentu pak může vypadat tak, že skript řídící simulaci bude opakovaně spouštět simulaci pro pevný počet cyklů a po každém jejím zastavení zkoumat stav systému.

V tomto případě však existuje problém se zablokováním běhu skriptu, dokud všechny procesy nezastaví. Skript se musí zastavit do doby, než dostane informaci, že všechny procesy už provedly požadovaný počet cyklů a zastavily svou činnost. Rozhodl jsem se pro tyto účely vytvářet speciální procesy, jejichž jediným účelem bude sbírat od procesů reprezen-

tujících uzly aplikací informace o tom, že se zastavily. Úkolem těchto procesů je udržovat informace o počtu ještě běžících procesů reprezentujících uzly a případně poskytovat informace o tom, že se všechny uzly již zastavily – čekání skriptu pak bude možno realizovat pasivně, skript prostě bude vyčkávat na speciální zprávu o zastavení všech procesů.

5.2 Popis implementace

Jak už bylo zmíněno, framework pro modelování gossipových systému je realizován v jazyce Erlang. Samotný framework je implementován v modulu `gossip` (dostupný v souboru `gossip.erl`). Framework je doplněn o modul `peersample` (v souboru `peersample.erl`), který poskytuje obecnou implementaci aplikace Peer sampling. Na základě této implementace je možné relativně snadno vystavět konkrétní variantu aplikace Peer sampling.

5.2.1 Modul gossip

Modul `gossip` poskytuje rozhraní pro vytváření modelů gossipových systému a pro simulační experimenty s těmito modely.

Konkrétní model gossipového systému je vhodné vložit do zvláštního modulu. Tento modul musí exportovat všechny funkce, které slouží k popisu aplikací.

Způsob popisu aplikace

Gossipový systém je popsán jako seznam aplikací. Aplikaci je možné popsat dvěma způsoby. Typický způsob popisu chápe danou aplikaci jako zcela běžnou aplikaci Gossipu odpovídající schématu z části 3.1. V tom případě je aplikace popsána sedmicí (tedy jako sedmiprvkový *tuple*) obsahující následující prvky (v tomto pořadí):

1. jméno dané aplikace (typu atom, každá aplikace musí mít v rámci daného modelu unikátní jméno)
2. funkce reprezentující update() s aritou 3, parametry jsou v tomto pořadí: stav daného uzlu, stav komunikačního partnera, stavy ostatních aplikací
3. funkce pro inicializaci uzlu s aritou 1, parametr je číslo daného uzlu
4. funkce reprezentující selectPeer() s aritou 3, parametry značí v tomto pořadí: stav daného uzlu, číslo daného uzlu, stavy ostatních aplikací
5. informace o variantě použitého protokolu Gossipu, možné hodnoty jsou `push`, `pull` a `pushpull` (pro varianty *push*, *pull* resp. *push-pull*)
6. funkce reprezentující wait() s aritou 2, parametry značí v tomto pořadí: aktuální stav daného uzlu, číslo daného uzlu
7. seznam jmen těch aplikací, od kterých tato aplikace přijímá stav (tj. které aplikace mají této aplikaci posílat svůj stav v rámci meziaplikační komunikace). Všechna jména v tomto seznamu musejí odpovídat jménům jiných aplikací přítomných v tomto systému. Jméno této aplikace nesmí být součástí seznamu (nedávalo by to smysl). Na pořadí jmen v seznamu nezáleží.

Na konkrétních jménech funkcí nezáleží. Číslo uzlu je vždy celé číslo v rozmezí $1, \ldots, n$, kde n je maximální počet uzlů v systému – toto číslo je specifikováno při vytváření systému.

Stavy ostatních aplikací, které přijímají funkce reprezentující update() a selectPeer(), jsou reprezentovány jako seznam dvojic obsahujících (v tomto pořadí) jméno aplikace a odpovídající stav. Tento seznam obsahuje dvojice jen pro ty cizí aplikace, které jsou uvedeny v poslední položce specifikace dané aplikace. Nicméně pořadí aplikací v těchto dvou seznamech si nemusí vzájemně odpovídat.

Dá se předpokládat, že v mnoha případech budou některé parametry předávané funkcím nadbytečné. Konkrétním realizacím funkcí potřebných pro popis aplikace samozřejmě nic nebrání některé parametry ignorovat. Musejí však vždy mít požadovanou aritu a dodržovat správné pořadí a formát neignorovaných parametrů.

Alternativní způsob popisu aplikace gossipu ji dovoluje uživateli popsat prostřednictvím funkcí active() a passive(), které definují chování aktivního a pasivního vlákna. V tomto případě je aplikace popsána pěticí (pětiprvkovým tuplem), jehož prvky obsahují v tomto pořadí:

1. jméno aplikace (typu atom, má totožnou funkci jako v předchozím případě)
2. funkce reprezentující active() s aritou 4, její parametry jsou v tomto pořadí: aktuální stav daného uzlu, číslo daného uzlu, stavy cizích aplikací, pravděpodobnost ztráty zprávy. Hodnota vracená touto funkcí je ignorována
3. funkce reprezentující passive() s aritou 4, její parametry jsou stejné jako v předchozím případě, tedy: aktuální stav daného uzlu, číslo daného uzlu, stavy cizích aplikací, pravděpodobnost ztráty zprávy. Funkce musí vrátit nový stav daného uzlu.
4. funkce pro inicializaci uzlu s aritou 1, jediný parametr značí číslo uzlu
5. seznam jmen aplikací, jejichž stav tato aplikace přijímá. Položka je totožná s poslední položkou v předchozím popisu.

Způsob, jak by se měly chovat funkce reprezentující active() a passive() byl popsán na straně 39. Stavy cizích aplikací a čísla uzlů mají stejné vlastnosti jako v předchozím případě.

Funkce active() a passive() také přijímají jako parametr pravděpodobnost ztráty zprávy. Jedná se o číslo typu float v intervalu $\langle 0, 1\rangle$ značící pravděpodobnost, s jakou funkce neodešle zprávu, kterou má odeslat (neodeslání zprávy simuluje její ztrátu). Kód popisující danou funkce by měl odesílat všechny (meziuzlové) zprávy pouze s danou pravděpodobností.

Je zřejmé, že popis aplikace takovýmto způsobem je velice nízkoúrovňový a tedy nepohodlný. Proto bych doporučoval používat popis aplikace podle schématu z 3.1, kdykoli je to možné.

Vytváření, ovládání a ukončování simulace

Nová simulace je vytvořena funkcí `gossip:newSimulation/4`. Parametry této funkce jsou v tomto pořadí: maximální počet uzlů v systému, seznam aplikací, jméno modulu a pravděpodobnost ztráty zprávy.

Formát seznamu aplikací byl popsán v předchozí podkapitole. Jméno modulu musí být jméno toho modulu, ve kterém jsou definovány všechny funkce popisující jednotlivé aplikace. Všechny tyto funkce také musejí být součástí rozhraní daného modulu.

Vytvořené procesy budou reprezentovány jménem aplikace a číslem uzlu. Čísla uzlu jsou v rozmezí $1, \ldots, n$, kde n je maximální počet uzlů v systému (určeno prvním parametrem funkce `newSimulation/4`).

Pravděpodobnost ztráty zprávy je reálné číslo v intervalu $\langle 0, 1 \rangle$, tento údaj slouží pro modelování ztrát zpráv.

Funkce `newSimulation/4` vytvoří všechny procesy tvořící danou simulaci – tedy jako procesy modelující jednotlivé uzly aplikací, tak procesy pro sbírání informací o ukončení běhu těchto uzlových procesů. Veškeré uzly aplikací budou po vytvoření „zapnuté“ a zastavené – tedy nebudou vyvíjet aktivní činnost.

Funkce `gossip:quitSimulation/2` slouží pro ukončení simulace. Její dva parametry jsou – počet uzlů a seznam aplikací – by měly být totožné s těmi, které dostala funkce `newSimulation/4`. Funkce `quitSimulation/2` provede zrušení všech procesů reprezentujících simulaci.

Pro komunikaci s uzly za běhu aplikace slouží následující funkce definované v rozhraní modulu `gossip`:

`getNodeState/3` – požádá daný uzel aplikace, aby vrátil svůj stav. Parametry jsou: jméno aplikace, číslo uzlu, délka čekání na odpověď. Poslední parametr je buď číslo v milisekundách, jak dlouho hodlá uživatel čekat na odpověď, nebo atom `infinity`, pokud hodlá čekat donekončena. Délka čekání by neměla být menší než předpokládaná průměrná délka cyklu. Funkce vrací dvojici: číslo uzlu a stav, je-li uzel zapnut; atom `switchedOff`, je-li uzel vypnut; nebo atom `noAnswer`, pokud uzel nestihl odpovědět.

`getSetOfNodesState/3` – požádá zadanou množinu uzlů aplikace, aby vrátily svůj stav. Parametry: jméno aplikace, seznam čísel uzlů (které hodlá žádat), délka čekání na odpověď (stejná jako v předchozím případě). Funkce vrací seznam stavů jednotlivých uzlů. Pokud některý uzel nestihne odpovědět, jeho stav nebude v seznamu zahrnut. Pořadí stavů v navraceném seznamu nemusí odpovídat pořadí čísel uzlů ve druhém parametru funkce. Funkce se chová efektivněji než opakované volání funkce `getNodeState/3`.

`getNodeSteps/3` – požádá daný uzel aplikace, aby vrátil informaci o tom, kolik cyklů hodlá provést, než se zastaví. Pokud vrátí hodnotu 0, uzel je již zastaven. Parametry jsou: jméno aplikace, číslo uzlu, délka čekání na odpověď. Funkce vrací dvojici: číslo uzlu a počet kroků, je-li uzel zapnut; atom `switchedOff`, je-li uzel vypnut; nebo atom `noAnswer`, pokud uzel neodpověděl.

`getNodeSwitchInfo/3` – požádá daný uzel aplikace, aby vrátil informaci o tom, zda je zapnutý nebo vypnutý. Parametry jsou: jméno aplikace, číslo uzlu, délka čekání na odpověď. Funkce vrací dvojici: číslo uzlu a atom `on` nebo `off`, stihl-li uzel odpovědět. V opačném případě vrací atom `noAnswer`.

`setAllSteps/3` – požádá všechny uzly aplikace, aby provedly zadaný počet cyklů. Parametry jsou: jméno aplikace, maximální počet uzlů v systému, počet kroků.

`setNodeSteps/3` – požádá daný uzel aplikace, aby provedl zadaný počet cyklů. Parametry jsou: jméno aplikace, číslo uzlu, počet cyklů.

`sendMsgToAllNodes/3` – zašla zadanou zprávu všem uzlům. Parametry jsou: jméno aplikace, maximální počet uzlů v systému, zpráva k odeslání. Zpráva k odeslání je jeden z těchto atomů:

- `switchOff` – přikáže uzlům, aby se vypnuly (pokud je uzel již vypnutý, bude ignorováno)
- `switchOn` – přikáže uzlům, aby se zapnuly (pokud je již zapnutý, bude ignorováno)
- `switchChange` – přikáže uzlům, aby se zapnuly, jsou-li vypnuty; nebo se vypnuly, jsou-li zapnuty
- `stop` – přikáže uzlům, aby se zastavily (uzly, které jsou již zastaveny, budou tuto zprávu ignorovat)
- `goon` – přikáže uzlům, aby pokračovaly ve výpočtu, dokud nebudou explicitně zastaveny (tj. provede potenciálně neomezený počet kroků)

`sendMsgToNode/3` – zašle zadanou zprávu danému uzlu. Parametry: jméno aplikace, číslo uzlu, zpráva k odeslání. Možné zprávy k odeslání jsou stejné jako u `sendMsgToAllNodes/3`.

`stopAllNodes/2` – volání `gossip:stopAllNodes(App, Nr)` je ekvivalentní k volání `gossip:sendMsgToAllNodes(App, Nr, stop)`. Přikáže tedy všem uzlům, aby se zastavily

`goonAllNodes/2` – volání `gossip:goonAllNodes(App, Nr)` je ekvivalentní k volání `gossip:sendMsgToAllNodes(App, Nr, goon)`. Přikáže tedy všem uzlům, aby se spustily.

`runNSteps/3` – požádá všechny uzly, aby provedly pevný počet cyklů a zablokuje běh aktuálního procesu, dokud všechny uzly nezastaví. Parametry jsou: jméno aplikace, počet uzlů v systému a počet cyklů, které se mají provést. Podstatné je, že tato funkce nechá provádět daný počet cyklů pouze uzly jedné aplikace. Je-li daná aplikace závislá na nějaké aplikaci jiné, je třeba tuto aplikaci před provedením `runNSteps/3` spustit (např. pomocí `sendMsgToAllNodes/3` s parametrem `goon`) a po provedení této funkce zase vypnout (např. `sendMsgToAllNodes/3` s parametrem `stop`).

Všechny tyto funkce je možné používat pouze po té, co byla simulace řádně vytvořena funkcí `newSimulation/4`, a před tím, než je simulace ukončena funkcí `quitSimulation/2`.

Vedle těchto funkcí poskytuje modul `gossip` pomocnou funkci `getProcessName/2`. Parametry funkce značí jméno aplikace a číslo uzlu, funkce vrací registrované jméno procesu reprezentujícího daný uzel dané aplikace. Podobně funkce `getSupervisorName/1` vrací jméno pomocného procesu pro sledování běhu uzlů zadané aplikace.

Reprezentace uzlu aplikace

Běh procesu pro sledování zastavení běhu uzlů je popsán funkcí `supervisorRun/2`. Běh procesu reprezentujícího uzel aplikace je popsán funkcí `nodeRun/8`. Tato funkce velmi dobře odpovídá pseudokódu ze strany 39. Funkce `nodeRun/8` pracuje v koncové rekurzi, je tedy praktičtější vnímat její opakovaný běh jako cyklus. Funkce tedy opakovaně volá reprezentace funkcí active() (`active/6`), passive() (`passive/6`), a funkce `userMsgs/4`, `interAppRecv/2` a `interAppSend/4`. Jediná z těchto funkcí, která provádí čekání na příchozí zprávy, je funkce `passive/6`. Ostatní funkce pouze volají kód, případně vybírají zprávy ze schránky s nulovou dobou čekání.

Funkce `active/6` a `passive/6` reprezentují chování aktivního a pasivního procesu. Pokud je aplikace Gossipu popsána jako pětice, tj. prostřednictvím reprezentace funkcí active()

a passive(), provádějí funkce `active/6` a `passive/6` prosté provedení odpovídajících reprezentací funkcí popisujících chování systému. Veškerá meziuzlová komunikace je v tomto případě v režii uživatele.

V případě, když je aplikace Gossipu popsána podle kostry Gossipu, tj. prostřednictvím reprezentací funkcí selectPeer(), update(), wait(), atd., provádějí funkce `active/6` a `passive/6` kód, kterým simulují chování dané kostry gossipové aplikace. V tomto případě probíhá meziuzlová komunikace přesně podle popisu v této kostře. Specifikace varianty Gossipu (tj. *push*, *pull* nebo *push-pull*) ovlivňuje pouze aktivní část popisu chování uzlu, tedy pouze určuje, jaké typy zpráv bude uzel aplikace odesílat. Pasivní část popisu chování uzlu je ochotná reagovat vždy na všechny typy zpráv, bez ohledu na použitou variantu Gossipu (tedy např. aplikace využívající variantu *pull* bude schopná reagovat i na zprávy typu *push*). Toho lze s úspěchem využít při realizaci jednoduché možnosti zásahů vnějšího světa do běžícího modelu. Reakce uzlu aplikace na různé typy zpráv meziuzlové komunikace jsou popsány ve funkci `gossipMsg/5`.

Funkce `userMsgs/4` zpracovává příchozí uživatelské zprávy. Uživatelské zprávy jsou zprávy, které používá uživatel (v případě interaktivní simulace to je přímo uživatel, v případě spouštění popisu simulace v nějakém skriptu se jedná o ten skript) pro ovládání běhu simulace a zjišťování jejího stavu. Tato funkce při každém svém volání zpracuje všechny uživatelské zprávy, které se v průběhu cyklu ve schránce nashromáždily. Funkce neprovádí čekání na příchozí zprávy. To, že je funkce volána jen jednou za cyklus a je jedinou funkcí, která příchozí zprávy zpracovává, způsobuje určité zpoždění při reakci uzlu na uživatelskou zprávu (průměrná doba zpoždění je polovina cyklu).

V rámci uživatelských zpráv je možno zapínat, vypínat, zastavovat a spouštět uzel aplikace reprezentovaný daným procesem. Dále je možné získávat informace o stavu daného uzlu aplikace a o stavu běhu uzlu – tedy informaci, jestli je uzel vypnut či zapnut, zastaven či spuštěn, případně kolik cyklů má v úmyslu provést, než se zastaví.

V implementaci je rozdíl mezi vypnutím (resp. zapnutím) a zastavením (resp. spuštěním) uzlu ten, že proces reprezentující zastavený uzel pouze přestal provádět funkci `active/6`. Tedy všecny ostatní funkce (tedy `passive/6`, `userMsgs/4`, `interAppRecv/2` a `interAppSend/2`) jsou pořád opakovaně prováděny. Uzel pak negeneruje žádnou vlastní aktivitu, ale je schopen reagovat na zprávy z vnějšku.

Proces reprezentující vypnutý uzel tento cyklus neprovádí, pouze čeká na přijetí zprávy požadující jeho zapnutí či oznamující konec simulace (na ostatní uživatelské zprávy reaguje pouze oznámením, že je vypnut). Po zapnutí se proces vrací přesně do toho stavu, ve kterém byl předtím, než byl vypnut.

Funkce `interAppRecv/2` a `interAppSend/4` slouží pro zpracování meziaplikačních zpráv. Funkce `interAppRecv/2` provádí přijetí příchozích meziaplikačních zpráv. Funkce zpracuje všechny meziaplikační zprávy, které proces během cyklu obdržel, avšak neprovádí čekání na přichozí zprávy – pouze vybere ze schránky ty, které v průběhu cyklu přišly. Funkce `interAppSend/4` provede rozeslání meziaplikačních zpráv odpovídajícím uzlům všech aplikací, které od této aplikace tyto zprávy přijímají. Toto rozeslání se tedy provádí právě jednou za cyklus.

5.2.2 Modul peersample

Modul `peersample` reprezentuje obecnou implementaci aplikace Peer sampling. Ve svém rozhraní poskytuje několik funkcí, které je možné snadno využít pro sestavení konkrétní realizace této aplikace. Podstatné je, že tento modul nelze využít přímo jako kompletní

aplikaci.

Modul poskytuje ve svém rozhraní čtyři funkce:

`getPeer/2` – funkce slouží pro výběr náhodného komunikačního partnera pro konkrétní uzel konkrétní aplikace. Požaduje dva parametry: jméno aplikace, *ve které bude využita* (tedy funkční aplikace, která využívá aplikaci Peer sampling), a aktuální stav aplikace Peer sampling. Funkce vrací registrované jméno procesu jakožto komunikačního partnera volajícího uzlu. Funkce by měla být volána z funkce selectPeer() té funkční aplikace, která využívá aplikaci Peer sampling. Jméno aplikace zadané jako první parametr by tedy mělo být jméno volající aplikace.

`selectPeer/3` – funkce slouží pro sestavení funkce selectPeer() konkrétní realizace aplikace Peer sampling. Parametry jsou: jméno aplikace realizující Peer sampling (tedy jméno té aplikace, ze které je volána), aktuální stav dané aplikace a specifikace varianty realizace této funkce. Specifikace varianty funkce selectPeer() je jeden ze dvou atomů:

- `random` – jako komunikační partner bude vybrán náhodný prvek z aktuálního pohledu
- `last` – bude vybrán poslední prvek z aktuálního pohledu

Funkce vrací adresu vybraného komunikačního partnera.

`update/3` – funkce slouží pro sestavení funkce update() konkrétní realizace aplikace Peer sampling. Parametry jsou: stav uzlu, stav komunikačního partnera a specifikace varianty realizace této funkce. Specifikace varianty funkce update() je jeden ze dvou atomů:

- `random` – při sestavování nového stavu dojde k náhodnému výběru deskriptorů
- `first` – při sestavování nového stavu dojde k výběru deskriptorů s nejnižšími hodnotami *hop count*

Funkce vrací nový stav sestavený z obou stavů dodaných jako vstupy.

`init/4` – funkce slouží pro sestavení funkce pro inicializaci stavu prvků. Požadované parametry jsou: jméno konkrétní realizace aplikace Peer sampling, číslo tohoto uzlu, celkový počet uzlů v systému, maximální velikost pohledu aplikace Peer sampling.

Význam možných variant funkcí selectPeer() a update() je popsán v části 4.2.

Stav uzlu aplikace Peer sampling je zde definován jako čtveřice (tedy čtyřprvkový *tuple*) obsahující tyto prvky:

1. aktuální pohled daného uzlu
2. aktuální velikost pohledu
3. číslo uzlu
4. maximální velikost pohledu

Aktuální velikost pohledu je udržována čistě z praktických důvodů (aby nebylo nutné velikost pohledu neustále přepočítávat). Číslo uzlu a maximální velikost pohledu jsou neměnné části stavu.

Aktuální pohled daného uzlu je reprezentován jako seznam dvojic sestávajících z čísla uzlu a odpovídající hodnoty *hop count* (její význam je popsán v části 4.2). Tento seznam je podle této hodnoty vzestupně uspořádán.

Nicméně z hlediska uživatele není definice stavu podstatná – uživatel se může dívat na stav konkrétní realizace aplikace Peer sampling jako na „černou skříňku“, jejíž obsah ho nezajímá.

Příklad použití

Z předchozího popisu nemusí být zcela zřejmý způsob použití výše popsaného rozhraní pro definici konkrétní varianty aplikace Peer sampling.

Předpokládejme, že chceme sestavit variantu aplikace Peer sampling, jejíž funkce update() provádí náhodný výběr nové množiny deskriptorů a jejíž funkce selectPeer() vrací vždy poslední prvek z aktuálního stavu. Pojmenujme tuto aplikaci atomem `ps`.

Předpokládejme dále, že požadovaná maximální velikost pohledu bude 40 a celkový počet uzlů bude 1000. Funkce wait() bude vždy požadovat dobu čekání 500 ms.

Realizaci funkce update() této aplikace pojmenujeme `psUpdate/3` a nadefinujeme ji triviálně jako:

```
psUpdate(State, PeersState, _) ->
  peersample:update(State, PeersState, random).
```

Funkce ignoruje třetí parametr, který ji simulátor Gossipu předá – stavy ostatních aplikací.

Realizaci funkce selectPeer() pojmenujeme `psSelectPeer/3` a nadefinujeme ji jako:

```
psSelectPeer(State, _, _) ->
  peersample:selectPeer(ps, State, last).
```

První parametr předaný funkci `peersample:selectPeer/3` je jméno této realizace aplikace Peer sampling, poslední parametr zase značí, že má provádět výběr náhodného prvku.

Realizaci funkce pro inicializaci stavu uzlu můžeme pojmenovat `psInit/1` a nadefinovat:

```
psInit(NodeNumber) ->
  peersample:init(ps, NodeNumber, 1000, 40).
```

Třetí parametr značí maximální počet uzlů v systému (musí se jednat o stejnou hodnotu, která byla předána funkci `gossip:newSimulation/4`). Poslední parametr značí maximální velikost pohledu. V praxi bývá samozřejmě výhodnější tyto konstanty definovat prostřednictvím konstantního makra preprocesoru nebo jako výsledek konstantní funkce.

Funci wait můžeme realizovat jako

```
psWait(_, _) -> 500.
```

Funkce ignoruje oba své parametry a vrací konstantu.

Danou realizaci aplikace Peer sampling poté můžeme popsat jako sedmici

```
{ps, psUpdate, psInit, psSelectPeer, pushpull, psWait, []}
```

Aplikace Peer sampling je samozřejmě typu *push-pull* (tudíž pátý prvek je atom `pushpull`) a není závislá na žádné jiné aplikaci (proto je posledním prvkem prázdný seznam).

Funkce `peersample:getPeer/2` je určena pro volání z funkce selectPeer() jiné aplikace, která využívá služeb naší realizace aplikace Peer sampling. Nechť se tato cizí aplikace jmenuje `someOtherApp` a nechť se její realizace funkce selectPeer() jmenuje `soaSelectPeer/3`. Definice této funkce může vypadat jako:

```
soaSelectPeer(_, _, [{ps, PeerSampleState}|_]) ->
  peersample:getPeer(someOtherApp, PeerSampleState);
soaSelectPeer(_, _, [_|Rest]) -> soaSelectPeer(ignored, ignored, Rest).
```

Funkce ignoruje své první dva parametry, které jí předá simulátor Gossipu – tj. ignoruje stav své aplikace i číslo svého uzlu a bere v úvahu pouze seznam stavů ostatních aplikací. Protože neznáme předem umístění stavu aplikace `ps` v seznamu stavů přijímaných aplikací (tj. v třetím parametru funkce `soaSelectPeer` reprezentující selectPeer() této aplikace), je nutné procházet postupně celý seznam tak dlouho, dokud hledaný stav nenajdeme.

5.2.3 Implementace úloh

Pomocí naimplementovaného frameworku jsem vytvořil modely dříve popsaných gossipových systémů. Modely jsou určeny převážně na interaktivní činnost v reálném čase, nicméně existuje zde možnost jejich použití v dávkově zapsaných simulacích.

Každý model je implementován v rámci jednoho modulu Erlangu. Většina těchto modulů má velice podobné rozhraní – toto rozhraní v podstatě jen zapouzdřuje obecné rozhraní pro práci s procesy reprezentujícími uzly aplikací poskytované modulem `gossip`.

Společné rozhraní modulů `aver1`, `aver2`, `infodiss` a `topo` (ostatní moduly implementují jen část tohoto rozhraní):

`create/1` – vytvoří daný model a nastaví mu pravděpodobnost ztráty zprávy (jediný argument; musí být v intervalu $\langle 0, 1 \rangle$). Model bude po vytvoření ve stavu zastaveno.

`create/0` – ekvivalentní k `create(0)`.

`stop/0` – zastaví běh simulace

`goon/0` – (znovu)spustí běh simulace na neurčitou dobu

`goon/1` – (znovu)spustí běh simulace, nechá všechny uzly provést počet cyklů zadaný jediným argumentem a zastaví

`quit/0` – ukončí simulaci

`runNSteps/1` – nechá všechny uzly v systému provést zadaný počet cyklů a zablokuje běh volajícího procesu, dokud všechny uzly nezastaví. Tato funkce je vhodná zejména pro použití ve skriptech, není příliš vhodná pro interaktivní simulaci

`nodeNr/0` – vrací počet uzlů v systému

`getState/1` – vrátí stav uzlu, jehož číslo bylo zadáno jako argument, toto číslo musí být v rozmezí $1, \ldots,$`nodeNr()`.

`getSteps/1` – vrací počet cyklů, které hodlá provést uzel, jehož číslo bylo zadano jako parametr, než se zastaví

Vedle tohoto společného rozhraní poskytují některé moduly ve svém rozhraní i další funkce, které jsou specifické pro aplikaci, která je v daném modulu naimplementována. Tyto zvláštnosti jsou popsány v následujících odstavcích.

Modul `aver1`

Jedná se o jednoduchý model aplikace Gossipu provádějící agregaci dat – výpočet průměru náhodně zvolených hodnot.

Systém se skládá z jediné aplikace – agregace dat. Aplikace Peer sampling zde není přítomna, její funkčnost je nahrazena pouhým náhodným výběrem čísla uzlu komunikačního partnera s rovnoměrným rozložením pravděpodobnosti v rozsahu $1, \ldots,$`nodeNr()`.

Stav uzlu je dán reálným číslem. Jednotlivé uzly jsou inicializovány náhodným číslem s rovnoměrným rozložením pravděpodobnosti v rozsahu $0, \ldots, 100$. Systém by měl konvergovat k průměru, který je roven přibližně hodnotě 50.

Modul `aver2`

Jedná se o komplikovanější variantu modelu aplikace Gossipu provádějící výpočet průměru náhodných hodnot.

Systém se skládá ze dvou aplikací: agregace dat a Peer sampling. Aplikace agregace dat se od varianty z modulu `aver1` liší jen tím, že její jednotlivé uzly jsou inicializovány náhodným číslem s rovnoměrným rozložením pravděpodobnosti v rozmezí $1, \ldots, 1000$ a tím, že využívá pro výběr komunikačního partnera služby Peer sampling.

Vedle výše popsaného obecného rozhraní poskytuje modul ještě následující funkce:

`switchInfo/1` – vrací informaci, jestli je uzel, jehož číslo bylo zadáno jako parametr, vypnut či zapnut

`changeSwitch/1` – je-li uzel, jehož číslo bylo zadáno jako parametr, vypnut, bude zapnut, je-li zapnut, bude vypnut

`switchOff/1` – vypne uzel, jehož číslo bylo zadáno jako parametr (byl-li uzel již vypnut, bude ignorováno)

`switchOn/1` – zapne uzel, jehož číslo bylo zadáno jako parametr (byl-li uzel již zapnut, bude ignorováno)

Modul `infodiss`

Jedná se o model gossipové aplikace realizující šíření informací (*information dissemination*).

Systém se skládá ze dvou aplikací: šíření informací a Peer sampling. Stav aplikace pro šíření informací je tvořen databází. Tato databáze je zde reprezentována jako seznam trojic `{Key, Val, Timestamp}`, kde `Key` je klíč záznamu, `Val` je hodnota přiřazená tomuto klíči a `Timestamp` je časová známka poslední změny tohoto záznamu.

Součástí rozhraní je vedle dříve zmíněného obecného rozhraní také funkce `addKeyVal/4`, která slouží pro zadávání změn obsahu distribuované databáze. Význam argumentů funkce (v tomto pořadí): klíč vkládaného záznamu, hodnota vkládaného záznamu, časová známka, číslo uzlu, u kterého bude daná změna databáze provedena. Časová známka je celé číslo reprezentující čas provedené změny databáze (důležité je, aby změnám provedeným později byla přiřazena vyšší časová známka, s reálným časem nemusí mít tato hodnota nic společného). Pokud v databázi daného uzlu už daná hodnota klíče je, bude hodnota přiřazená tomuto klíči přepsána jen tehdy, je-li časová známka této změny větší než časová známka předchozího záznamu.

Modul `mcast`

Jedná se o model aplikace realizující pravděpodobnostní multicast. Systém se skládá ze dvou aplikací: pravděpodobnostní multicast a Peer sampling.

Stav uzlu aplikace pravděpodobnostní multicast je tvořen seznamem přijatých zpráv. Zpráva je zde typicky atom.

Podstatnou vlastností pravděpodobnostního multicastu je, že provádí činnost i v případě, je-li jeho běh zastaven. Veškerá jeho činnost je totiž implementována v rámci funkce passive(), funkce active() neprovádí nic. V důsledku toho není možné činnost této aplikace zastavit (tedy převést do stavu zastaveno). Tato nevýhoda je dána samou podstatou práce této aplikace.

Modul implementuje celé obecné společné rozhraní s výjimkou funkce `runNSteps/1`, která (vzhledem k tomu, že běh hlavní aplikace není možné zastavit) u této aplikace nedává příliš velký smysl. Funkce `stop/0`, `goon/0`, `goon/1` a `getSteps/1` ovlivňují pouze aplikaci Peer sampling, na funkčnost pravděpodobnostního multicastu nemají přímý vliv (nicméně nefunkční aplikace Peer sampling samozřejmě pravděpodobnostní multicast ovlivní).

Z předchozího plyne, že tuto aplikaci není možné příliš dobře využít ve skriptech popisujících simulační experimenty.

Součástí rozhraní je funkce `sendToNode/2`, která slouží pro zaslání zprávy od uživatele nějakému uzlu. Tato zpráva bude následně rozšířena mezi všechny uzly systému. Parametry této funkce jsou v tomto pořadí: číslo uzlu a zaslaná zpráva (typicky typu atom).

Modul `superpeer`

Jedná se o model aplikace pro sestavení superpeerové topologie. Systém se skládá ze čtyř aplikací: samotné sestavení superpeerové topologie a tří variant aplikace Peer sampling. Tyto aplikace byly popsány v části 4.3.

Uzly této aplikace generují činnost i jsou-li zastaveny. Klienti i superpeery aktivně počítají cykly, ve kterých nedostali zprávu od svých protějšků (klienti od svého superpeeru, superpeery od svých klientů), klienti v případě překročení určité meze aktivně žádají jiné superpeery o zařazení do jejich množin klientů, superpeery v případě překročení této meze aktivně odstraňují klienty ze svých množin klientů. V tomto systému tudíž nedává smysl možnost zastavení jeho běhu. Rozhraní je v důsledku toho značně jednoduché:

`create/1` – vytvoří novou simulaci a spustí ji. Jediný parametr je číslo *speed-up*, které představuje průměrné zrychlení části aplikace udržující vazby mezi klienty a superpeery oproti části aplikace provádějící přesun klientů k superpeerům s vyšší kapacitou.

`create/2` – obdobné jako `create/1`, první parametr je stejný jako v předchozím případě, druhý parametr je atom představující název souboru, na jehož konec budou během simulace vypisovány statistické informace o průběhu simulace. Pokud je jako druhý parametr zadána hodnota jiného typu než atom, bude ignorována.

`quit/0` – ukončí simulaci (bez ohledu na to, jestli byla vytvořena příkazem `create/1` nebo `create/2`)

`getState/1` – vypíše na standardní výstup stav všech aplikací uzlu, jehož číslo bylo zadáno jako jediný parametr

`stats/0` – vypíše informace o počtu superpeerů a klientů v systému, průměrnou kapacitu superpeerů a součet velikostí množin klientů všech superpeerů

`statsBMS/0` – vypíše statistiky o počtu zatím zaslaných zpráv *becomeMySuperpeer*.

Jednotlivé uzly jsou inicializovány takto: kapacita je vybrána jako náhodné nezáporné číslo z Gaussova rozložení pravděpodobnosti se střední hodnotou 1 a variancí 5, počáteční role uzlu (klient nebo superpeer) je vybrána náhodně se stejnou pravděpodobností pro obě hodnoty, množina klientů superpeeru je zpočátku prázdná, jako počáteční superpeer klienta je vybrán náhodný uzel bez ohledu na jeho roli.

Hodnota *speed-up* (požadovaná funkcemi `create/1` a `create/2`), tedy průměrné zrychlení té části aplikace *Superpeer topology*, která slouží k udržování vazeb mezi klienty a superpeery, oproti té části, která slouží k přesunu klientů k superpeerům s vyšší kapacitou, je nutná pro stabilitu systému. Ukázalo se totiž, že systém, tak jak byl popsán v části 4.3, je silně nestabilní, topologie, kterou generuje, neodpovídá požadavkům na superpeerovou topologii a ani zdaleka nekonverguje k nějakému stabilnímu uspořádání. Problém byl v tom, že přenos klientů mezi superpeery probíhal příliš rychle a systém nebyl schopen ustálit vztah mezi klienty a superpeery.

Ukázalo se dále, že chování této implementace je značně výpočetně náročné a silně závislé na výpočetním výkonu počítače, na kterém je prováděno. Aplikace může s jistým parametrem *speed-up* vykazovat na jednom počítači vlastnosti odpovídající požadavkům (tedy sestavovat kvalitní superpeerovou topologii) a na počítači jiném s tím samým parametrem vykazovat chování spíše chaotické.

Paradoxní je, že tato aplikace, jakožto aplikace gossipová (v širším významu), by měla teoreticky vykazovat chování, které je značně odolné vůči chybám a poruchám. Z teoretického popisu v části 4.3 se zdá být intuitivně jasné, že tato aplikace bude vždy sestavovat superpeerovou topologii s minimálním počtem superpeerů. Nicméně při experimentech s touto implementací jsem zjistil, že aplikace je značně citlivá na nastavení parametru *speed-up* a na vlastnostech prostředí, ve kterém je spouštěna, v případě nevhodných parametrů vykazuje spíše chaotické chování. Tudíž v tomto případě rozhodně nebylo dosaženo požadované odolnosti a robustnosti.

Pro rozumné chování systému na daném počítači je potřeba najít vhodnou hodnotu parametru *speed-up*. Ta je závislá na výpočetním výkonu a vytížení počítače v momentě, kdy je aplikace spouštěna. Při určování vhodné hodnoty *speed-up* pro daný počítač je vhodné experimentovat s hodnotami na exponenciální škále od hodnoty 10 až po několik set.

Druhý parametr funkce `create/2` slouží k zadání jména souboru, na jehož konec budou v průběhu simulace vkládány informace o statistických vlastnostech simulovaného systému – počet superpeerů a klientů v systému, součet velikostí množin *ClientSet* všech superpeerů (tato hodnota by ve správně se chovajícím systému měla vždy přibližně odpovídat počtu klientů v systému), průměrná kapacita superpeeru a informace o počtu odeslaných zpráv *becomeMySuperpeer* od začátku běhu systému. K tomuto výpisu dochází (v případě nepřetíženého systému) každých pět sekund bez ohledu na počet cyklů provedených simulovaným systémem. Jednotlivé výpisy jsou očíslovány a označeny hodnotou *nilProb*, což je hodnota získaná odvozená z hodnoty *speed-up* podle vzorce $1 - \frac{1}{\textit{speed-up}}$ (pro simulátor je přirozenější pracovat s touto upravenou hodnotou).

Popis aplikace v části 4.3 i tato implementace jsou postaveny na předpokladu, že množina *underloaded* (tj. stav aplikace *Underloaded sampling*) obsahuje vždy dostatečně přesné informace o nevytížených superpeerech. Funkce `statsBMS/0` slouží k ověření, jestli byl tento předpoklad správný. Tato funkce využívá zkutečnosti, že množina *underloaded* je využívána k výběru parametru pro zprávu *becomeMySuperpeer*. Funkce vypisuje statistiku o počtu zpráv *becomeMySuperpeer*, které přijali klienti, nevytížené superpeery a vytížené

superpeery.

Z experimentů, které jsem provedl, vyplývá, že (i v případě vhodně nastavené hodnoty *speed-up*) je pouze přibližně desetina zpráv typu *becomeMySuperpeer* přijímána nevytíženými superpeery. Ostatních devět desetin zpráv tohoto typu je přijímáno vytíženými superpeery nebo dokonce klienty. Tudíž předpoklad, že množina *underloaded* obsahuje s vysokou pravděpodobností správné údaje, byl nesprávný. To také může vysvětlovat překvapivě špatné chování této aplikace.

Modul `topo`

Model aplikace pro sestavení topologie dvourozměrné mřížky.

Systém se skládá ze dvou aplikací: sestavení topologie a Peer sampling. Stav aplikace pro sestavení topologie je tvořen pětic `{View, ViewSize, XVal, YVal, Addr}`, kde `View` je pohled daného prvku, `ViewSize` je velikost tohoto pohledu, `XVal` a `YVal` jsou souřadnice x a y tvořící profil daného uzlu, `Addr` je jméno jeho procesu. Pohled je tvořen uspořádaným seznamem trojic `{Addr, XVal, YVal}` popisujících cizí uzel.

Součástí rozhraní modulu je vedle obecného rozhraní také funkce `getAvgMetric/1`, která provádí výpočet průměrné vzdálenosti zadaného uzlu od uzlů nalézajících se v jeho pohledu.

5.2.4 Známé problémy

Během experimentování s modely vytvořenými ve frameworku se u některých procesů modelujících uzly aplikace objevil problém se značně velkým množstvím položek ve schránce přijatých zpráv – byly pozorovány případy, kdy schránka procesu obsahovala i několik set zpráv, které proces nevyzvedl. Tento problém se týká pouze velmi malého množství procesů – naprostá většina procesů modelujících uzly aplikace měla po většinu doby výpočtu ve schránce řádově jednotky zpráv.

Tento problém byl pozorován u modulů `superpeer` a `aver2`. Oba tyto moduly provádějí výpočetně značně náročnou činnost – v obou případech byl v době pozorování tohoto jevu procesor plně vytížen.

U modulu `aver2` byl tento jev pozorován poté, co byl běh simulace spuštěn na neurčitou dobu příkazem `goon/0`. Jakmile byl běh simulace zastaven, schránky zpráv všech procesů se vyprázdnily (včetně procesů, které předtím měly ve schránce stovky zpráv). Při opakovaném spouštění běhu simulace na omezený počet cyklů např. příkazem `goon/1` nebo `runNSteps/1` k tomuto jevu zřejmě nedochází – při každém zastavení běhu simulace dojde k vyprázdnění všech schránek.

Zdá se tedy, že k tomuto hromadění zpráv ve schránce nedochází v důsledku toho, že je daný proces nevybírá, ale v důsledku toho, že proces není spouštěn dostatečně často v důsledku přetížení běhu procesoru při nepřerušovaném běhu simulace. Nemohu vyloučit, že k tomuto jevu dochází v důsledku nějaké chyby či nevhodného chování virtuálního stroje Erlangu. Jev byl pozorován u virtuálního stroje BEAM verze 5.5.5.

Problémem u modulu `superpeer` je ten, že jeho běh nelze smysluplně zastavit. V důsledku toho nelze u tohoto modulu toto hromadění zpráv ve schránce nějak omezit.

Přehled o množství zpráv ve schránkách jednotlivých procesů je možné získat v Erlang shellu například pomocí příkazu `i()`. Informaci o počtu zpráv ve schránce konkrétního procesu je možné získat pomocí standardní vestavěné funkce `process_info/2`, jejíž první parametr je identifikátor daného procesu a druhý parametr je atom `message_queue_len`.

5.3 Experimenty s frameworkem

S vytvořenými modely gossipových aplikací jsem provedl několik simulačních experimentů. Hlavním smyslem těchto experimentů není ani tak získat nové informace o daných aplikacích (tím se zabývají práce, ve kterých byly tyto aplikace definovány), jako ověřit a demonstrovat možnosti praktického použití vytvořeného frameworku.

Modul `experiments` obsahuje popis provedených experimentů. Jeho rozhraní poskytuje dvě funkce: `average/3` a `dissem/3`.

5.3.1 Rychlost konvergence agregace dat

Funkce `average/3` slouží pro provedení simulačního experimentu s modelem popsaným v modulu `aver2` pro různé pravděpodobnosti ztráty zpráv a výpadku uzlu. Jedná se o aplikaci provádějící průměrování. Systém se skládá z 1000 uzlů inicializovaných náhodnými hodnotami vybranými s rovnoměrným rozložením pravděpodobnosti v rozmezí $1, \ldots, 1000$. Systém je považován za zkonvergovaný, pokud je variance hodnot 100 náhodně vybraných uzlů menší než 0,02.

Funkce `average/3` požaduje následující argumenty: pravděpodobnost ztráty zprávy (reálné číslo v intervalu $\langle 0, 1 \rangle$), pravděpodobnost výpadku uzlu (v intervalu $\langle 0, 1 \rangle$), jméno souboru, do kterého bude uložen výsledek experimentu (typu atom). Funkce provede pro zadané hodnoty pět experimentů a jejich výsledky vypíše jak na standardní výstup, tak i do zadaného souboru. Hodnoty vypisované do souboru budou připojeny na jeho konec, předchozí obsah souboru ovlivněn nebude. Na standardní výstup jsou vedle výsledků dokončených experimentů vypisovány i informace o průběhu experimentu právě prováděného.

Do souboru jsou vypisovány výsledky experimentů v následujícím formátu. Výsledek každého experimentu je vypsán na jeden řádek. Ten začíná slovem `averaging` následovaným parametry experimentu: pravděpodobností ztráty zpráv (reálné číslo v intervalu $\langle 0, 1 \rangle$), pravděpodobností selhání uzlu (v intervalu $\langle 0, 1 \rangle$) a číslem experimentu (v rámci pěti experimentů provedených za sebou, číslovány od nuly). Poslední hodnota je výsledek experimentu – počet cyklů nutných pro konvergenci systémů.

Při provádění experimentu jsou prováděny vždy tři cykly výpočtu naráz. Po jejich provedení se zkontroluje, jestli systém zkonvergoval. Výsledný počet cyklů, které systém potřeboval ke své konvergenci, je tedy zaokrouhlen na nejbližší vyšší celočíselný násobek tří.

Výsledky experimentů – počty cyklů nutných pro konvergenci systému pro různé pravděpodobnosti ztráty zpráv při nulové pravděpodobnosti selhání uzlu:

Ztráty zpráv	0 %	10 %	20 %	30 %	40 %	50 %
Průměrný počet cyklů	15,0	18,0	21,0	24,0	27,6	33,0

Výsledky experimentů při pevně nastavené pravděpodobnosti ztráty zpráv 10 % pro různé pravděpodobnosti selhání uzlů:

Selhání uzlu	0 %	10 %	20 %	30 %	40 %	50 %
Průměrný počet cyklů	18,0	30,0	38,4	46,2	54,0	64,2

V obou případech jsou vypsané hodnoty průměrného počtu provedených cyklů nutných pro konvergenci systému získány jako průměr výsledků pěti experimentů provedených s daným nastavením pravděpodobnosti ztráty zpráv a výpadku uzlů.

5.3.2 Rychlost šíření informací

Funkce `dissem/3` slouží pro provedení simulačního experimentu s modelem aplikace pro šíření informací popsaným v modulu `infodiss`. Parametry jsou stejné jako v předchozím

případě, tedy: pravděpodobnost ztráty zpráv, pravděpodobnost výpadku uzlů, jméno souboru, na jehož konec budou připojeny výsledky experimentů. Funkce pro zadané hodnoty provede pět experimentů a jejich výsledky – počty cyklů nutných k rozšíření informace po celém systému – vloží do zadaného souboru. Funkce během provádění experimentu vypisuje na standardní výstup informace o právě prováděné činnosti.

Systém se skládá z 1000 uzlů, každý uzel má na počátku prázdnou databázi. Po vytvoření modelu je jednomu uzlu vložena do databáze právě jedna zpráva. Tato zpráva je považována za plně rozšířenou, pokud je obsažena v databázích 100 náhodně vybraných uzlů. Je vždy prováděn pouze jeden cyklus výpočtu naráz, po provedení daného cyklu je zkontrolováno, jestli se daná zpráva již rozšířila.

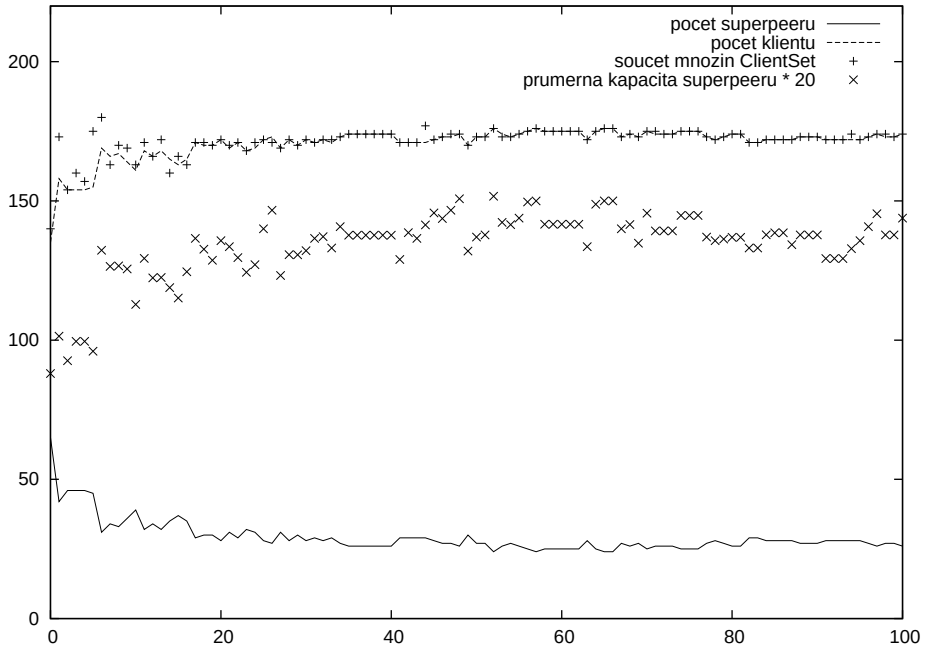
Formát dat vkládaných do souboru je následující. Výsledek každého experimentu je vložen na jeden řádek, ten začíná slovem `dissemination`, za ním jsou parametry experimentu: pravděpodobnost ztráty zprávy a pravděpodobnost selhání uzlu, následuje číslo experimentu v rámci pěti experimentů provedených za sebou (číslováno od nuly) a výsledek experimentu.

Výsledky experimentů při nulové pravděpodobnosti selhání uzlu a různých pravděpodobnostech ztráty zpráv:

Ztráty zpráv	0 %	10 %	20 %	30 %	40 %	50 %	60 %	70 %
Průměrný počet cyklů	6,0	6,0	6,0	6,8	6,8	7,0	26,2	30,8

Vypsané výsledky jsou vždy průměr hodnot získaných z pěti experimentů provedených pro danou pravděpodobnost ztráty zpráv.

Zajímavý je zejména prudký nárůst průměrného počtu cyklů nutných pro rozšíření informace mezi 50 a 60 procenty ztracených zpráv.



Obrázek 5.2: Výsledky simulačního experimentu s modulem `superpeer`

5.3.3 Experimenty s modulem `superpeer`

Jak už bylo zmíněno, chování aplikace pro sestavení superpeerové topologie popsané v části 4.3 neodpovídá zcela chování očekávanému.

Abych prezentoval chování této aplikace, předkládám zde výsledky jednoho simulačního experimentu provedeného na mém počítači (obrázek 5.2).

Experiment byl proveden s nastavením *speed-up* 40. Jednotka na časové ose odpovídá 5 sekundám vývoje systému. Celkový počet uzlů v systému je 200. Zobrazeny jsou informace o počtu klientů a superpeerů, o součtech velikostí množin $ClientSet$ a o průměrné kapacitě superpeeru (tato hodnota je pro lepší názornost vynásobena dvaceti).

Systém vykazuje některé předpokládané vlastnosti: růst počtu klientů na úkor počtu superpeerů a růst průměrné kapacity superpeerů. Na druhou stranu se nezdá, že by systém konvergoval ke zcela stabilní topologii.

Kapitola 6

Závěr

V této práci jsem určil sebeorganizaci a emergenci jako velice podstatné koncepty pro návrh komplexních *self-** počítačových systémů. Poskytuji podrobný rozbor obou fenoménů včetně popisu jejich podstatných vlastností a příkladů jejich výskytu.

Vedle emergence a sebeorganizace se tato práce zabývá také generickým protokolem Gossip, který popisuje jako prostředek pro návrh systémů využívajících emergenci a sebeorganizaci. Práce dále obsahuje popis několika aplikací tohoto protokolu včetně popisu jejich implementace.

Hlavní částí této práce je framework sloužící pro modelování a simulaci systémů postavených na Gossipu. Framework je implementován v jazyce Erlang. Umožňuje vytváření modelů vhodných jak pro interaktivní práci v reálném čase, tak i určených pro simulační experimenty řízené skriptem. Jednou z výhod tohoto frameworku je, že model v něm vytvořený je tvořen souběžně běžícími procesy, umožňuje tedy simulaci provádět paralelně.

V tomto frameworku bylo také vytvořeno několik modelů dříve popsaných gossipových systémů a s těmito modely byly provedeny simulační experimenty.

Framework je možné používat pro modelování a simulaci decentralizovaných systémů popsaných množinou rovnocenných uzlů, jejichž chování využívá fenoménů emergence a sebeorganizace. Je vhodný zejména pro modelování systémů, pro které platí, že chování jejich uzlů odpovídá obecné kostře Gossipu v tom pojetí, v jakém byla popsána v této práci. Vedle toho se však tento framework dá použít i pro systémy s chováním značně odlišným.

6.1 Možnosti rozšíření

Určité možnosti rozšíření tohoto frameworku zcela jistě existují. Oblast systémů skládajících se z velkého množství rovnocenných uzlů je značně rozsáhlá a pestrá, nebylo tudíž možné předem určit potřebné vlastnosti prostředku schopného modelovat všechny takovéto systémy. Navíc gossipové systémy nemají pevně danou definici – jako gossipové bývají v literatuře běžně označovány i systémy s výrazně odlišnými vlastnostmi, než byly popsány v této práci. Snadno se tak může stát, že uživatel bude chtít vytvořit model, který přesahuje možnosti současné podoby tohoto frameworku.

Jednou z nevýhod tohoto frameworku jsou omezené možnosti synchronizace běhu procesů představujících jednotlivé uzly aplikací. Ta je v současné době možná pouze prostřednictvím zastavení běhu všech těchto procesů naráz, případně prostřednictvím provedení konstantního počtu cyklů všemi těmito procesy. Synchronizovaný běh všech procesů je možné do určité míry simulovat pouze pomocí opakovaného provádění jediného cyklu naráz

všemi procesy.

Jako jedna z možností rozšíření tohoto frameworku se tedy nabízí možnost provést rozšíření jeho schopnosti synchronizovat běh jednotlivých procesů představujících uzly aplikace.

Literatura

[1] Armstrong, J.: Concurrency Oriented Programming in Erlang. 2003.

[2] Armstrong, J.: *Programming Erlang.* The Pragmatic Bookshelf, 2007.

[3] Armstrong, J.; Virding, R.; Wikström, C.; aj.: *Concurrent Programming in Erlang.* Prentice Hall, druhé vydání, 1996, ISBN 0-13-508301-X.

[4] Babaoglu, O.: P2P Computing, Autonomic Computing and Gossip-based Techniques. 2008, učební text.

[5] Babaoglu, O.; Jelasity, M.: Self-* properties through gossiping. *Philosophical Transactions of the Royal Society*, ročník 366, 2008.

[6] Bar-Yam, Y.: A Mathematical Theory of Strong Emergence Using Multiscale Variety. *Complexity*, ročník 9, č. 6, 2004.

[7] Breddin, I.: Self-Organization and Emergence. *Organic Computing*, 2006.

[8] Cesarini, F.; Thompson, S.: *Erlang Programming.* O'Reilly, první vydání, 2009, ISBN 978-0-596-51818-9.

[9] Correia, L.: Self-organised systems: fundamental properties. *Revista de Ciências da Computação*, ročník I, 2006.

[10] De Wolf, T.; Holvoet, T.: Emergence and Self-Organisation: a statement of similarities and differences. *Engineering Self-Organising Systems*, 2005.

[11] Demers, A.; Greene, D.; Hauser, C.; aj.: Epidemic algorithms for replicated database maintenance. *In Proc. 6th Annual ACM Symposium on Principles of Distributed Computing (PODC'87)*, 1987: s. 1–12.

[12] Erlang/OTP System Documentation. 2011, [online], [cit. 2011-04-25].
URL http://www.erlang.org/doc/pdf/otp-system-documentation.pdf

[13] Eugster, P.; Felber, P.; Le Fessant, F.: The Art of Programming Gossip-based Systems. *ACM SIGOPS Operating Systems Review – Gossip-based computer networking archive*, ročník 41, č. 5, 2007.

[14] Halley, J. D.; Winkler, D. A.: Classification of Emergence and its Relation to Self-Organization. *Complexity*, 2008.

[15] Jelasity, M.: Engineering Emergence through Gossip. *Proc. AISB Convention, Joint Symposium on Socially Inspired Computing*, 2005: s. 123–126.

[16] Jelasity, M.; Babaoglu, O.: T-Man: Fast Gossip-based Construction of Large-Scale Overlay Topologies. 2004, technical report UBLCS-2004-7.

[17] Jelasity, M.; Guerraoui, R.; Kermarrec, A. M.; aj.: The Peer Sampling Service: Experimental Evaluation of Unstructured Gossip-Based Implementations. *Middleware*, 2004, lNCS 3231.

[18] Kermarrec, A.; Massoulié, L.; Ganesh, A. J.: Probabilistic Reliable Dissemination in Large-Scale Systems. *IEEE Transactions on Paralel and Distributed Systems*, ročník 14, č. 3, 2003.

[19] Logan, M.; Merritt, E.; Carlsson, R.: *Erlang and OTP in Action*. Manning, 2011, ISBN 9781933988788.

[20] M. Jelasity, O. B., A. Montresor: A Modular Paradigm for Building Self-Organizing Peer-to-Peer Applications. *Engineering Self-Organising Systems*, ročník 2977/2004, 2004: s. 265–282, dOI: 10.1007/978-3-540-24701-2-18.

[21] Montressor, A.: A Robust Protocol for Building Superpeer Overlay Topologies. 2004, technical report UBLCS-2004-8.

Dodatek A

Obsah DVD

Součástí práce je i DVD obsahující tento dokument v elektronické podobě, zdrojové kódy tohoto dokumentu pro systém LaTeX a zdrojové kódy vytvořeného frameworku včetně implementace několika modelů.

Soubor `xkunst01.pdf` obsahuje tento dokument v elektronické podobě ve formátu PDF. Kompletní zdrojové kódy tohoto dokumentu včetně `Makefile` se nalézají ve složce `latex/`.

Složka `aplikace/` obsahuje zdrojové kódy modulů tvořících framework (v souborech `gossip.erl` a `peersample.erl`), implementace jednotlivých aplikací (soubory `aver1.erl`, `aver2.erl`, `mcast.erl`, `infodiss.erl`, `superpeer.erl` a `topo.erl`) a implementaci experimentů (soubor `experiments.erl`). Všechny tyto soubory byly popsány v částech 5.2 a 5.3. Složka obsahuje také soubor `Makefile` využívající kompilátor Erlangu `erlc`.

Složka `experimenty` obsahuje textové soubory s výsledky experimentů popsaných v části 5.3. Soubor `aver.experiment` obsahuje výsledky několika experimentů získaných funkcí `average/3` modulu `experiments`, soubor `dissem.experiment` obsahuje výsledky experimentů získaných funkcí `dissem/3` téhož modulu. Soubor `superpeer-sp40.experiment` obsahuje výsledky experimentu s modulem `superpeer` provedeného na mém počítači s nastavením *speed-up* 40.

Dodatek B

Softwarové požadavky

Framework je určen pro jazyk Erlang. Implementace virtuálního stroje tohoto jazyka jsou k dispozici pro naprostou většinu dnes používaných operačních systémů včetně MS Windows, Linux, FreeBSD a OS X. Systém Erlang/OTP včetně dokumentace je možné získat na adrese `http://www.erlang.org/`.

Framework byl testován na implementaci virtuálního stroje BEAM verze 5.5.5.

V prostředí CVT FIT je v současné době virtuální stroj Erlangu BEAM dostupný na serveru Eva pod příkazem `erl`.

Dodatek C

Formát zpráv modulu `gossip`

Následuje popis formátu zpráv zasílaných mezi procesy tvořícími model gossipového systému.

Znalost tohoto formátu může být pro uživatele užitečná například v případě, kdy potřebuje debugovat jím naimplementovaný model gossipového systému, nebo v případě modelování možnosti zásahu do systému z vnějšího prostředí.

Obecně mají všechny zasílané zprávy tvar n-tice (dvojice až pětice), jejíž první položka obsahuje identifikátor procesu odesilatele a druhá položka je atom určující typ zprávy. Počet a význam ostatních položek této n-tice je závislý na typu zprávy. Počet a význam ostatních položek této n-tice je závislý na typu zprávy.

C.1 Zprávy zajišťující meziuzlovou komunikaci

Tyto zprávy jsou používány pouze aplikacemi, které byly zadány prostřednictvím reprezentací funkcí update(), selectPeer(), wait(), atd., tedy aplikacemi odpovídajícími obecné kostře Gossipu (vizte 3.1).

V případě aplikací zadaných prostřednictvím reprezentací funkcí active() a passive() je zajištění meziuzlové komunikace zcela v režii uživatele a je jen na něm, jestli využije tohoto formátu zpráv, nebo si definuje formát vlastní.

Obecný formát zpráv meziuzlové komunikace je `{PID, MsgType, State}`, kde `PID` je identifikátor procesu odesilatele, `State` je zasílaný stav a `MsgType` je typ zasílané zprávy:

`push` – zpráva zasílaná v případě varianty Gossipu *push*. Pokud proces přijme tuto zprávu, nebude na ni odpovídat.

`pull` – zpráva zasílaná v případě varianty Gossipu *pull*. Stav zaslaný společně s touto zprávou je příjemcem ignorován. Proces, který přijme tuto zprávu, na ni odpoví zprávou typu `pullansw`.

`pullansw` – zpráva zasílaná v případě varianty Gossipu *pull* jako odpověď na zprávu typu `pull`.

`pushpull` – zpráva zasílaná v případě varianty Gossipu *push-pull*. Proces, který přijme tuto zprávu, na ni odpoví zprávou typu `pushpullansw`.

`pushpullansw` – zpráva zasílaná v případě varianty Gossipu *push-pull* jako odpověď na zprávu typu `pushpull`.

Ačkoli uzly aplikace Gossipu zasílají pouze zprávy odpovídající jejich variantě Gossipu (tedy uzly varianty *push* zasílají jen zprávy typu `push`, atd.), jsou všechny uzly ochotny reagovat na jakýkoli typ zprávy bez ohledu na svou variantu Gossipu. Tohoto faktu (tedy např., že uzly varianty *push-pull* jsou ochotny reagovat i na zprávy typu `push`) lze využít například při implementaci možnosti zásahů do systému z vnějšku.

C.2 Zprávy zajišťující meziaplikační komunikaci

Existuje pouze jeden typ zprávy pro meziaplikační komunikaci. Formát této zprávy je `{PID, interApp, AppName, NodeNr, State}`, kde `PID` je identifikátor procesu, který tuto zprávu odesílá, `AppName` je jméno aplikace odesílatele, `NodeNr` je číslo uzlu odesílatele (musí být totožné s číslem uzlu příjemce), `State` je zasílaný aktuální stav odesílatele.

C.3 Zprávy pro komunikaci mezi uživatelem a procesem

Prostřednictvím těchto zpráv zasílá uživatel příkazy procesům reprezentujícím uzly aplikací modelovaného systému.

Obecný tvar těchto zpráv je `{PID, MsgType}`, kde `PID` je identifikátor procesu odesílatele (tedy typicky procesu shellu nebo procesu provádějícího skript realizující simulační experiment) a `MsgType` je typ odeslané zprávy. Tyto zprávy jsou zasílané funkcemi z modulu `gossip`. Některé typy zpráv přímo odpovídají typům zpráv, které očekávají na svém vstupu funkce `gossip:sendMsgToAllNodes/3` a `sendMsgToNode/3` (byly popsány v části popisující implementaci modulu `gossip`). Vedle těchto typů zpráv tady jsou i typy používané jinými funkce z tohoto modulu.

`switchOff` – přikáže uzlu (reprezentovanému procesem příjemce), aby se vypnul

`switchOn` – přikáže uzlu, aby se zapnul

`switchChange` – přikáže uzlu, aby změnil svůj stav (tj. je-li vypnut, aby se zapnul, je-li zapnut, aby se vypnul)

`switchInfo` – příjemce této zprávy zašle jejímu odesilateli zprávu typu `switchansw` s informací, jestli je uzel vypnut nebo zapnut

`state` – příjemce této zprávy zašle jejímu odesilateli zprávu typu `stateansw` s informací o aktuálním stavu uzlu aplikace, který je tímto procesem reprezentován. Pokud je uzel vypnut, zašle odesilateli zprávu typu `switchedOffError`.

`steps` – příjemce této zprávy zašle jejímu odesilateli zprávu typu `stepsansw` s informací o počtu cyklů, které má daný proces provést, než se zastaví. Pokud je uzel vypnut, zašle odesilateli zprávu typu `switchedOffError`.

`stop` – přikáže uzlu, aby se okamžitě zastavil

`goon` – přikáže uzlu, aby pokračoval v činnosti (po neomezenou dobu)

`quit` – přikáže uzlu, aby ukončil svou činnost (odesíláno pouze na konci simulace)

Vedle těchto zpráv do této kategorie patří i zpráva ve tvaru `{PID, step, N}`. Ta slouží pro zaslání příkazu, aby proces provedl přesně `N` cyklů a zastavil se.

Zprávy zasílané procesem uživateli jako odpověď na požadavky tímto uživatelem zaslané mají tvar `{PID, MsgType, Answ}`, kde `Answ` je odpověď na daný požadavek a `MsgType` je typ zprávy:

`stateansw` – odpověď na zprávu `state`, `Answ` obsahuje zasílaný aktuální stav

`stepsansw` – odpověď na zprávu `steps`, `Answ` obsahuje počet cyklů, které má proces provést, než se zastaví (pokud proces už stojí, jedná se o hodnotu `0`, pokud má proces běžet donekonečna, jedná se o hodnotu `infinity`)

`switchansw` – odpověď na zprávu `switchInfo`, `Answ` obsahuje hodnotu `on` nebo `off` podle toho, jestli je uzel zapnut nebo vypnut

Zpráva typu `switchedOffError`, sloužící k upozornění uživatele, že požaduje informaci o stavu nebo počtu cyklů od uzlu, který je vypnut, má tvar `{PID, switchedOffError}`.

C.4 Zprávy pro komunikaci s procesy kontrolujícími běh systému

V systému jsou obecně dva typy procesů: procesy reprezentující uzly aplikace a procesy sloužící pro udržování informací o počtu zastavených a běžících procesů dané aplikace. Následující zprávy slouží pro komunikaci s druhou kategorií procesů.

C.4.1 Zprávy od uživatele

Jedná se o zprávy `{PID, quit}` a `{PID, await, Nr}`. Tyto zprávy jsou zasílány uživatelem procesu kontrolujícímu běh systému. Zpráva `quit` slouží pro ukončení činnosti daného procesu při ukončování simulace.

Zpráva `await` oznamuje procesu, že má očekávat přesně `Nr` zpráv typu `stopped` nebo `switchedOffError` od různých procesů reprezentujících uzly aplikace. Poté, co daný proces všechny tyto zprávy obdrží, odešle procesu uživatele zprávu typu `allStopped`. Tento mechanismus slouží pro implementaci funkce `gossip:runNSteps/3`.

C.4.2 Zprávy od uzlů aplikace

Jedná se o zprávy, které danému procesu zasílají procesy reprezentující uzly aplikace. Zpráva tvaru `{PID, stopped, MyNr}` je automaticky zaslána pokaždé, když daný proces reprezentující uzel aplikace zastaví.

Zpráva tvaru `{PID, switchedOffError}` je procesu kontrolujícímu běh systému zaslána tehdy, když vypnutý uzel obdrží zprávu typu `step` (tedy aby provedl zadaný počet kroků a zastavil se). Stejně pojmenovaná zpráva slouží i pro komunikaci mezi uzlem aplikace a uživatelem (tato zpráva obecně slouží pro reakci na pokus uživatele komunikovat s procesem reprezentujícím vypnutý uzel).

C.4.3 Zprávy zasílané uživateli

Zpráva tvaru `{PID, allStopped, App}` slouží pro oznámení uživateli, že všechny procesy reprezentující uzly dané aplikace `App` zastavily svůj běh.